Declarative, Formal, and Extensible Syntax Definition for AspectJ

5

ABSTRACT

Aspect-Oriented Programming (AOP) is attracting attention from both research and industry, as illustrated by the ever-growing popularity of AspectJ, the *de facto* standard AOP extension of Java. From a compiler construction perspective, AspectJ is interesting as it is a typical example of a *compositional language*, i.e. a language composed of a number of separate languages with different syntactic styles: in addition to plain Java, AspectJ includes a language for defining pointcuts and one for defining advices. Language composition represents a non-trivial challenge for conventional parsing techniques. First, combining several languages with different lexical syntax leads to considerable complexity in the lexical states to be processed. Second, as new language features for AOP are being explored, many research proposals are concerned with *further extending* the AspectJ language, resulting in a need for an extensible syntax definition.

This chapter shows how *scannerless parsing* elegantly addresses the issues encountered by conventional techniques when parsing AspectJ. We present the design of a modular, extensible, and formal definition of the lexical and context-free aspects of the AspectJ syntax in the Syntax Definition Formalism SDF, which is implemented by a scannerless, generalized LR parser (SGLR). We introduce *grammar mixins* as a novel application of SDF's modularity features, which allows the declarative definition of different keyword policies and combination of extensions. We illustrate the modular extensibility of our definition with syntax extensions taken from current research on aspect languages. Finally, benchmarks show the reasonable performance of scannerless generalized LR parsing for this grammar.

5.1 INTRODUCTION

"A language that is used will be changed" to paraphrase Lehman's first law of software evolution [Lehman 1980]. Lehman's laws of software evolution apply to programming languages as they apply to other software systems. While the rate of change is high in the early years of a language, even standardized languages are subject to change. The Java language alone provides good examples of a variety of language evolution scenarios. Language designers do not get it right the first time around (e.g. enumerations and generics). Programming patterns emerge that are so common that they can be supported directly by the language (annotations, `foreach`, crosscutting concerns). The environ-

ment in which the language is used changes and poses new requirements (e.g. JSP for programming dynamic webpages). Finally, modern languages tend to become conglomerates of languages with different styles (e.g. LINQ, E4X, and the embedding of XML in XJ).

The *risks* of software evolution, such as reduced maintainability, understandability, and extensibility, apply to language evolution as well. While experiments are conducted with the implementation, the actual language definition diverges from the documented specification, and it becomes harder to understand what the language is. With the growing complexity of a language, further improvements and extensions become harder and harder to make. These risks especially apply to language conglomerates, where interactions between language components with different styles become very complex.

AspectJ [Kiczales et al. 2001], the *de facto* standard aspect-oriented programming language, provides a good case in point. While the official `ajc` compiler for AspectJ extends the mainstream Eclipse compiler for Java and has a large user base, the aspect-oriented paradigm is still actively being researched; there are many proposals for further improvements and extensions (e.g. [Masuhara & Kawauchi 2003, Sakurai et al. 2004, Bodden & Stolz, Tanter et al. 2006, Allan et al. 2005, Ongkingco et al. 2006, Harbulot & Gurd 2006]). The AspectBench Compiler `abc` [Avgustinov et al. 2005] provides an alternative implementation that is geared to experimentation with and development of new aspect-oriented language features.

AspectJ adds support to Java for modularization of crosscutting concerns, which are specified as separate *aspects*. Aspects contain *advice* to modify the program flow at certain points, called *join points*. AspectJ extends Java with a sub-language for expressing *pointcuts*, i.e. predicates over the execution of a program that determine when the aspect should apply, and *advices*, i.e. method bodies implementing the action that the aspect should undertake. The pointcut language has a syntax that is quite different from the base language. This complicates the parsing of AspectJ, since its lexical syntax is *context-sensitive*. This is a problem for scanners, which are oblivious to context. The parsers of the `ajc` and `abc` compilers choose different solutions for these problems. The `abc` parser uses a stateful scanner [Hendren et al. 2004], while the `ajc` compiler uses a handwritten parser for parsing pointcut expressions. For both parsers the result is an *operational*, rather than declarative, implementation of the AspectJ syntax, in particular the lexical syntax, for which the correctness and completeness are hard to verify, and that is difficult to modify and extend.

In this chapter, we present a declarative, formal, and extensible syntax definition of AspectJ. The syntax definition is *formal and declarative* in the sense that *all* aspects of the language are defined by means of grammar rules. The syntax definition is *modular and extensible* in the sense that the definition consists of a series of modules that define the syntax of the 'component' languages separately. AspectJ is defined as an extension to a syntax definition of Java 5, which can and has been further extended with experimental aspect features.

We proceed as follows. First we give brief introductions to concepts of

parsing (Section 5.2) and aspect-oriented programming (Section 5.3). To explain the contribution of our approach we examine in Section 5.4 the issues that must be addressed in a parser for AspectJ and discuss how the parser implementations of `ajc` (Section 5.5) and `abc` (Section 5.6), two state-of-the-art compilers for AspectJ, solve these issues. In Section 5.8 we present the design of a syntax definition for AspectJ that defines its lexical as well as context-free syntax, overcoming these issues. Our AspectJ syntax definition is based on the syntax definition formalism SDF2 [Visser 1997b] and its implementation with scannerless generalized LR parsing (SGLR) [Visser 1997a, van den Brand et al. 2002]. The combination of scannerless [Salomon & Cormack 1989, Salomon & Cormack 1995] and generalized LR [Tomita 1985] parsing supports the full class of context-free grammars and integrates the scanner and parser. Due to these foundations, the definition elegantly deals with the extension and embedding of the Java language, the problems of context-sensitive lexical syntax, and the different keyword policies of `ajc` and `abc`. For the latter we introduce *grammar mixins*, a novel application of SDF's modularity features.

In Section 5.9 we examine the extensibility of `ajc` and `abc` and we show how grammar mixins can be used to create and combine extensions of the declarative syntax definition. In Section 5.10 we discuss the performance of our implementation. While LALR parsing with a separate scanner is guaranteed to be linear in the length of the input, the theoretical complexity of generalized LR parsing depends on the grammar [Rekers 1992]. However, obtaining a LALR grammar is often a non-trivial task and context-sensitive lexical syntax further complicates matters. The benchmark compares the performance of the parser generated from our syntax definition to `ajc`, `abc`, and ANTLR. We conclude with a discussion of previous, related, and future work. In particular, we analyze why SGLR is not yet in widespread use, and discuss research issues to be addressed to change this.

The contributions of this chapter are:

- An in-depth analysis of the intricacies of parsing AspectJ and how this is achieved in mainstream compilers, compromising extensibility;
- A declarative and formal definition of the context-free *and* lexical syntax of AspectJ;
- A modular formalization of keyword policies as applied by the `ajc` and `abc` AspectJ compilers;
- An account of the application of scannerless parsing to elegantly deal with context-sensitive lexical syntax;
- A demonstration of the extensibility of our AspectJ syntax definition;
- A mixin-like mechanism for combining syntactic extensions and instantiating sub-languages for use in different contexts;
- A case study showing the applicability of scannerless generalized LR parsing to complex programming languages.

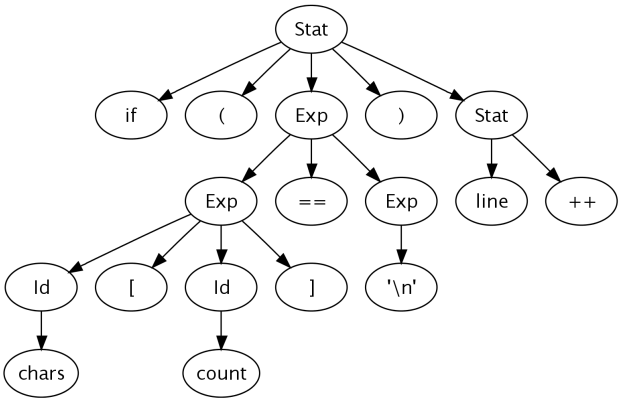

Figure 5.1 Simplified parse tree for a Java statement

AVAILABILITY The AspectJ syntax definition and parser are available as part of AspectJ-front, which is open source (LGPL) and available at `http://aspectj.syntax-definition.org`.

5.2 SCANNING AND PARSING

In this section we review the basic concepts of the conventional parser architecture using a separate scanner for tokenization and compare it to *scannerless* parsing, in which the parser reads characters directly. A parser transforms a list of characters (the program text) into a structured representation (a parse tree). For example, Figure 5.1 shows a (simplified) parse tree for the Java statement `if (chars[count] == '\n') line++;`. Parse trees are a better representation for language processing tools such as compilers than plain text strings.

5.2.1 *Tokenization or Scanning*

Conventional parsers divide the work between the proper parser, which recognizes the tree structure in a text, and a *tokenizer* or *scanner*, which divides the list of characters that make up the program text into a list of *tokens*. For example, the Java statement

```
if (chars[count] == '\n') line++;
```

is divided into tokens as follows

```
if ␣ ( chars [ count ] ␣ == ␣ '\n' ) ␣ line ++ ;
```

Figure 5.2 illustrates the collaboration between scanner and parser. The parser building a parse tree requests tokens from the scanner, which reads characters from the input string.

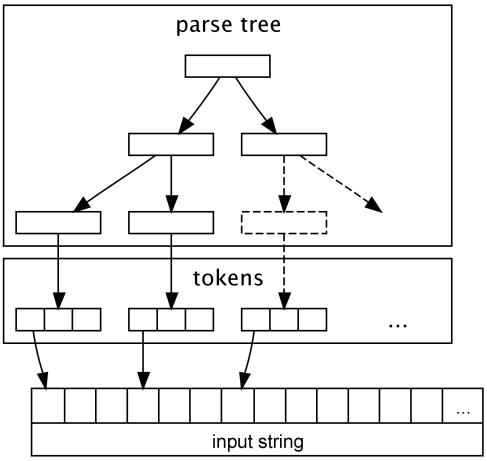

Figure 5.2 A scanner-based parser uses a scanner to partition the input string into tokens, which become the leafs of parse trees.

The reason for the division is the use of different techniques for the implementation of tokenization and parsing. Tokens can be recognized using a deterministic finite automaton (DFA), while parsers for recursive structures need a pushdown automaton, i.e. a stack. Furthermore, tokenization reduces the number of items the parser has to consider; long sequences of characters are reduced to a single token, and whitespace and comments are usually ignored by the parser.

5.2.2 *Scanner and Parser Generators*

Scanners and parsers can be generated from formal definitions of the lexical and context-free syntax of a language. Scanners are generated from regular expressions describing the tokens of the language and parsers are generated from context-free grammars (BNF). Conventional parser generators such as YACC, Bison, and CUP accept only a restricted class of context-free grammars such as LL, LR, or LALR. The advantage is that the complexity of parsers for such grammars is linear in the size of the input. Furthermore, grammars in these classes are not ambiguous; only one interpretation for any string is possible. The fact that a grammar does not belong in a certain class shows up as conflicts in the parse table. For example, in the case of LR parsing, shift-reduce and reduce-reduce conflicts indicate that the parser cannot make a decision about how to proceed based on the provided lookahead information. Solving such conflicts requires rewriting the grammar and sometimes changing the syntax of the language. Also, restricted classes of context-free grammars are not closed under composition.

5.2.3 *Lexical Context*

The uniform tokenization of the input string by means of regular expressions can be problematic, since the scanner does not consider the context in which a token occurs. This means that a particular sequence of characters is interpreted as the same token everywhere in the program text. For syntactically simple languages that have been designed with this restriction in mind this is not a problem. However, modern languages tend to become combinations of 'domain-specific' languages, each providing syntax appropriate for its domain. Because of the limitations of the ASCII character set, the same characters may play different roles in these languages.

One solution that is employed is the use of *lexical state*; the scanner operates in different modes depending on the context. This requires state switching when entering and leaving a context, and may require interaction between the scanner and the parser.

5.2.4 *Programmatic Parsers*

Another solution to bypass the restrictions posed by scanner and parser generators is the use of programmatic 'handwritten' parsers, usually according to the *recursive descent* (topdown parsing) approach. The advantage is that it is easy to escape the rigor of the parsing algorithm and customize it where necessary. Possible customizations are to drive tokenization from the parser to deal with lexical context or to provide customized error handling. A disadvantage of programmatic parsers is that a parser does not provide a declarative specification of the language; conversely, a formal grammar can serve as both documentation and implementation. Also, parser implementations are usually much larger in terms of lines of code, with all the implications for maintainability.

5.2.5 *Scannerless Generalized LR Parsing*

A *scannerless* parser does not make use of a separate scanner to tokenize the input [Salomon & Cormack 1995]; the parser directly reads the characters of the input string. Instead of a separate specification of lexical and context-free syntax as is customary in scanner-based parsing, a single grammar is used that defines all aspects of the language. Although there is no conceptual difference with scanner-based parsing, scannerless parsing is not in common use because it does not work with conventional parser generators. A grammar that describes the lexical as well as the context-free syntax of a language does not usually fit in the grammar classes supported by parser generators. The problem is that these algorithms need to make a decision on the first (few) token(s) in the input. In the case of scannerless parsing a decision may only be made after reading an unbounded number of characters. This problem is solved by the use of *Generalized LR* (GLR) parsing. GLR parsers use a parse table generated by a normal LR parser table generator, e.g. LALR(1) or SLR(1). At points in the input where the parser encounters a shift-reduce

or reduce-reduce conflict, there are multiple possible continuations. In that case a GLR parser simulates the execution of all possible LR parses in parallel. Scannerless GLR (SGLR) parsing adds a few disambiguation techniques to GLR parsing to make it suitable for scannerless parsing [Visser 1997a, Visser 1997b, van den Brand et al. 2002]. *Follow restrictions* define longest match disambiguation and *reject productions* express reserved word policies.

An advantage of SGLR parsing is that it deals naturally with the problem of lexical context. Rather than parsing a lexical entity in isolation, as is done with regular expressions, the parsing context acts naturally as lexical state. Thus, the same sequence of characters can be interpreted differently in different parts of a program.

In the following sections we closely examine the differences between scanner-based and scannerless parsing, by studying state-of-the-art implementations of parsers for AspectJ. In Section 5.4 we analyze the properties of the parsers of the `ajc` and `abc` compilers for AspectJ. In Section 5.8 we discuss a syntax definition for AspectJ using the declarative syntax definition formalism SDF2.

5.3 A QUICK INTRODUCTION TO ASPECTJ

To understand the examples and issues we discuss in this chapter, it is important to be somewhat familiar with the *syntactic structure* of an AspectJ program. This section briefly discusses the various constructs of AspectJ. (In this chapter we focus on the pointcut-advice mechanism of AspectJ.) Knowledge of their *semantics* is not necessary. For a more extensive account of the AspectJ language we refer to [AspectJ].

Figure 5.3 shows an AspectJ aspect for caching executions of the `calc` method of Fibonacci objects. It shows the concise syntax for defining pointcuts, an `around` advice, and how this is mixed with normal Java code (AspectJ keywords are in emphasized bold).

Aspect Declarations

Aspects can be declared, similar to Java classes, either as top-level entities or nested in Java classes [8]. An aspect declaration consists of a number of pointcut declarations and advices, as well as standard Java members (e.g. the `cache` field [9]).

Pointcuts

A pointcut is the specification of a pattern of join points of interest to a given aspect. Join points here are events in the dynamic execution of a program, e.g. a method call or an access to an object field. As such, the pointcut language of AspectJ is really a separate *domain-specific language* for identifying join points.

A pointcut is specified using a number of pointcut designators. Primitive pointcut designators refer to different kinds of operations in the execution of a program. For instance, `execution` refers to the execution of a method [11] — which method(s) is specified by giving a *method pattern* as explained below.

```
public aspect Caching { [8]
  private Map<Integer, Integer> cache = [9]
    new HashMap<Integer, Integer>();

  pointcut cached(int value): [10]
    execution(* Fib.calc(int)) && args(value); [11]

  int around(int value): cached(value) { [12]
    if(cache.containsKey(value)) {
      return cache.get(value);
    }
    else {
      int result = proceed(value); [13]
      cache.put(value, result);
      return result;
    }
  }
}
```

Figure 5.3 A sample caching aspect in AspectJ

Furthermore, some pointcut designators are used either to further restrict a pointcut, or to *bind* some values to pointcut formal parameters. In Figure 5.3, the pointcut is given a name (a *named pointcut*) and exposes one parameter of type `int` [10], which is bound via the `args` pointcut designator to the value of the argument to `calc` method executions [11].

Advice

Advice are pieces of code to execute when an associated pointcut matches. This piece of code, which is similar to a Java method body, can be executed before, after, or around the intercepted join point based on the *advice kind*. Since the caching aspect may actually replace the execution of `calc`, it is declared to be of the around kind [12]. As a consequence, its return type (`int`) must be specified. The caching advice is associated to the `cached` *named pointcut*, and it is parameterized by the value of the argument to `calc`. Within an around advice body, calling `proceed` results in the intercepted join point to be executed [13].

Patterns

Fundamental to the pointcut language of AspectJ are *patterns*. A *name pattern* is used to denote names (method names, field names, type names) in a declarative fashion, using wildcards such as `*` and `?`. A *type pattern* is used to denote types, e.g. `int` matches the primitive type `int`, while `A+` matches object type `A` and all its subtypes (`+`). A *method pattern* as in the `execution` pointcut designator in [11] identifies matching method signatures: a return type pattern (`*` in [11]), a declaring type pattern (`Fib` in [11]), a name pattern (`calc` in [11]), and then type patterns for the parameters (`int` in [11]).

In this section we give an overview of some of the challenges of parsing AspectJ. The overview is based on an analysis of the AspectJ language and a review of the source of the scanner and parser of the two major AspectJ implementations: the official AspectJ compiler `ajc`, and the `abc` compiler from the AspectBench Compiler project [Avgustinov et al. 2005]. The scanner and the parser of `abc` have partially been documented in [Hendren et al. 2004]. The purpose of this overview is to show that the parsers of the major implementations of AspectJ are not based on a declarative and complete definition of the language, which leads to minor differences between the two compilers and a lack of clarity about the exact language that each recognizes, as well as parsers that are not easy to maintain and extend.

The main source of the issues in parsing AspectJ is the difference between the lexical syntax of different parts of an AspectJ source file. Conventionally, parsers use a separate *scanner* (or *lexer*) for lexical analysis that breaks up the input character stream into a list of tokens, such as identifiers, literals, layout, and specific keywords such as `class`, `public`, and `try`. Usually this tokenization is applied uniformly to the text of a program, so at every offset in the input, all the same tokens of the language can be recognized by the scanner. However, this does not apply to AspectJ, which is in fact more like a mixture of three languages. Regular Java code, aspect declarations, and pointcut expressions each have a different lexical syntax.

For example, in Java, `get*` is an identifier followed by a multiplication operator, while in a pointcut expression it represents an *identifier pattern* that matches any identifier with prefix `get`. In the first case, the scanner should produce the tokens `get` `*`, while in the second case a single token `get*` would be expected. Similarly, the `+` in the pointcut `call(Foo+.new())` is not an addition operator, but a *subtype pattern* that matches any subclass of `Foo`. To complicate matters, Java code can also occur within a pointcut definition. For instance, the `if(...)` pointcut designator takes as an argument a plain Java expression.

The languages involved in AspectJ also have different *keywords*. Depending on the AspectJ implementation, these keywords might be *reserved* or not. For `ajc`, most keywords are not reserved, since at most places they are explicitly allowed as identifiers in the grammar. For example, `aspect` is a keyword, but it is allowed as the name of a local variable. Similarly, `around` is not allowed as the name of a method in an aspect declaration, but it is in regular Java code. On the other hand, `around` is allowed as the name of a local variable in regular Java as well as in aspect declarations. The `abc` compiler uses a different keyword policy. For example, `before` is a *keyword* in the context of an aspect declaration, but is an *identifier* in Java code *and* in pointcut expressions. In both compilers, pointcut expression keywords, such as `execution` and `get`, are allowed as elements of a pointcut name pattern, e.g. `Foo.execution` is a valid name pattern, and so is `get*`.

Hence, an AspectJ compiler needs to consider the *context* of a sequence of

characters to decide what kind of token they represent. Next, we discuss in detail how `abc` and `ajc` parse AspectJ.

5.5 THE AJC SCANNER AND PARSER

The official AspectJ compiler[1], `ajc`, extends the Eclipse compiler for Java, which is developed as part of the Eclipse Java Development Tools (JDT) [JDT Website]. The parser of `ajc` roughly consists of three components:

1. The *scanner* of `ajc` is a small extension of the regular Java scanner of the JDT. The JDT scanner and the extension in `ajc` are both written by hand. The scanner extension does nothing more than adding some keywords to the scanner.

2. The *parser* of `ajc` is generated from a grammar using the Jikes parser generator (these days also known as LPG, LALR Parser Generator). The grammar is a modified version of the JDT grammar for regular Java. It does not actually define the syntax of pointcut expressions: these are only scanned and parsed separately.

   The handwritten part of the JDT parser for constructing ASTs is extended as well. The original Java code has to be modified at some places, mostly for making the parser more flexible and extensible by introducing factory methods. Presumably, this could be merged with the JDT parser itself.

3. A handwritten recursive descent *pattern parser* is invoked to parse the pointcut expressions of AspectJ after the source file has been scanned and parsed by the previous components. Except for the `if` pointcut designator, the pattern parser works directly on the result of the `ajc` scanner, since the `ajc` parser parses pointcuts as a list of tokens.

5.5.1 *Parsing Pointcuts*

The most interesting part of the `ajc` parser is the handling of pointcuts.

Scanner

The `ajc` scanner is applied uniformly to the input program, which means that the same set of tokens is allowed at all offsets in the input. Note that the `ajc` scanner does not add tokens to the JDT scanner, except for some keywords, so the pointcuts are tokenized as any other part of the source file. For example, the pointcut of the caching aspect in Figure 5.3 is scanned to the following list of tokens:

```
execution ( * Fib .  calc ( int ) ) && args ( value )
```

This sequence of tokens is a correct tokenization of this pointcut, but our previous example of the simple name pattern `get*` is actually not scanned

[1] Our study is based on `ajc` version 1.5.0.

as the single token `get*`, but as the tokenization you would expect in the context of a regular Java expression: an identifier followed by a multiplication operator, i.e. the scanner produces the tokenization `get *`.

Still, this does not look very harmful, but actually scanning pointcuts and Java code uniformly can lead to very strange tokenizations. For example, consider the (somewhat artificial) pointcut `call(* *1.Function+.apply(..))`. For this pointcut the correct tokenization according to the lexical syntax of pointcuts is:

```
call ( * *1 . Function + . apply ( .. ) )
```

However, the `ajc` scanner produces the following list of tokens for this pointcut:

```
call ( * * 1.F unction + . apply ( . . ) )
```

Perhaps surprisingly, `Function` has been split up and the F is now part of the token `1.F`, which is a floating-point literal where the `F` is a floating-point suffix. Of course, a floating-point literal is not allowed at all in this context in the source file. As we will show later, the pattern parser needs to work around this incorrect tokenization.

Unfortunately, things can get even worse. Although rather uncommon, the first alpha-numerical character after the `*` in a simple name pattern can be a number (in fact, this is also the case in the previous floating-point example). The token that starts after the `*` will always be scanned as a number by the JDT scanner, and the same will happen in the `ajc` scanner. The JDT scanner checks the structure of integer and floating-point literals by hand and immediately stops parsing if it finds a token that should be a floating-point or integer literal according to the Java lexical syntax, but is invalid because certain parts of the literal are missing. This can result in error messages about invalid literals, while in this context there can never actually be a literal.

For example, scanning the pointcut `call(void *0.Ef())` reports an *"Invalid float literal number"* because the scanner wants to recognize `0.E` as floating-point literal, but the actual exponent number is missing after the exponent indicator `E`. As another example, scanning the pointcut `call(void Foo.*0X())` fails with the error message *"Invalid hex literal number"*, since `0X` indicates the start of a hexadecimal floating-point or integer literal, but the actual numeral is missing.

Parser

The `ajc` parser operates on the sequence of tokens provided by the scanner. Unfortunately, for pointcuts the parser cannot do anything useful with this tokenization, since it is not even close to the real lexical syntax of pointcuts in many cases. In a handwritten parser it might be possible to work around the incorrect tokenization, but the `ajc` parser is generated from a grammar using the Jikes parser generator. In a grammar workarounds for incorrect tokenizations are possible as well (as we will see later for parameterized types) but for pointcuts this would be extraordinarily difficult, if not impossible.

```
PointcutDeclaration ::=
  PcHeader FormalParamListopt ')' ':' PseudoTokens  ';'

DeclareDeclaration ::=
  DeclareAnnoHeader PseudoTokensNoColon ':' Anno ';'

PseudoToken ::=
  '(' | ')' | '.' | '*' | Literal | 'new'
  | JavaIdentifier | 'if' '(' Expression ')' | ...

ColonPseudoToken ::= ':'

PseudoTokens ::=
  one or more PseudoToken or ColonPseudoToken
PseudoTokensNoColon ::=
  one or more PseudoToken
```

Figure 5.4 Pseudo tokens in the `ajc` grammar for AspectJ

For these reasons, the parser processes pointcuts just as a list of tokens called *pseudo tokens* that are parsed separately by the handwritten *pattern parser*. In this way, the main parser basically just skips pointcuts and forwards the output of the scanner (with a twist for the `if` pointcut) to the pattern parser. It is essential that the parser can find the end of the pointcut without parsing the pointcut. Fortunately, this is more or less the case in AspectJ, since pointcuts cannot contain semicolons, colons, and curly braces, except for the expression argument of the `if` pointcut designator, which we will discuss later.

The handling of pointcuts using pseudo tokens is illustrated in Figure 5.4: the first production defines pointcut declarations, where the pointcut, recognized as a sequence of pseudo tokens, starts after the colon and is terminated by the semicolon. The second production for inter-type annotation declarations uses a somewhat smaller set of pseudo tokens, since it is terminated by a colon instead of a semicolon. Most of the Java tokens, except for curly braces, semicolons, and colons, but including keywords, literals, etc., are defined to be pseudo tokens.

The `if` pointcut designator is a special case, since it takes a Java expression as an argument. Of course, the pattern parser should not reimplement the parsing of Java expressions. Also, Java expressions could break the assumption that pointcuts do not contain colons, semicolons, and curly braces. For these reasons, the `if` pointcut designator is parsed by `ajc` parser as a special kind of pseudo token, where the expression argument is not a list of tokens, but a real expression node (see Figure 5.4).

Interestingly, this special pseudo token for the `if` pointcut designator reserves the `if` keyword in pointcuts, while all other Java keywords are allowed in name patterns. Hence, the method pattern `boolean *.if*(..)` is not allowed in `ajc` [2].

[2]This turned out to be a known problem, see bug 61535 in `ajc`'s Bugzilla: `https://bugs.eclipse.org/bugs/show_bug.cgi?id=61535`, which has been opened in May 2004.

Pattern Parser

Finally, the handwritten pattern parser is applied to pointcuts, which have been parsed as a sequence of pseudo tokens by the parser. The pattern parser takes a fair amount of code, since the pointcut language of AspectJ is quite rich. Most of the code for parsing pointcuts is rather straightforward, though cumbersome to implement by hand. The most complex code handles the parsing of name patterns. Since the tokenization performed by the `ajc` scanner is not correct, the pattern parser cannot just consume the tokens. Instead, it needs to consider all the possible cases of incorrect tokenizations. For example, the pointcuts `call(* *1.foo(..))` and `call(* *1.f oo(..))` are both tokenized in the same way by the `ajc` scanner:

```
call ( * * 1.f oo ( . . ) ) ;
```

However, the token sequences for these two pointcuts cannot be handled in the same way, since the second one is incorrect, so a parse error needs to be reported. Therefore, the pattern parser checks if tokens are adjacent or not:

```
while(true) {
  tok = tokenSource.peek();
  if(previous != null) {
    if(!isAdjacent(previous, tok))
      break;
  }
  ...
}
```

The need for this adjacency check follows naturally from the fact that the pattern parser has to redo the scanning at some parts of the pointcut and a single AspectJ pointcut token can span multiple Java tokens, in particular in name patterns.

The special `if` pseudo tokens do not have to be parsed anymore. For this purpose, the `IToken` interface, of which `PseudoToken` and `IfPseudoToken` are implementations, is extended with a method *maybeGetParsedPointcut* that immediately returns the pointcut object. This method is invoked from the pattern parser:

```
public Pointcut parseSinglePointcut() {
  IToken t = tokenSource.peek();
  Pointcut p = t.maybeGetParsedPointcut();
  if(p != null) {
    tokenSource.next();
    return p;
  }
  String kind = parseIdentifier();
  ... // continue parsing the pointcut
}
```

5.5.2 *Parameterized Types*

Incorrect tokenization problems are not unique to AspectJ. Even in regular Java 5, a parser that applies a scanner uniformly to an input program has to

```
TypeArgs ::= '<' TypeArgList1

TypeArgList  -> TypeArg
TypeArgList  ::= TypeArgList ',' TypeArg
TypeArgList1 -> TypeArg1
TypeArgList1 ::= TypeArgList ',' TypeArg1
TypeArgList2 -> TypeArg2
TypeArgList2 ::= TypeArgList ',' TypeArg2
TypeArgList3 -> TypeArg3
TypeArgList3 ::= TypeArgList ',' TypeArg3

TypeArg  ::= RefType
TypeArg1 -> RefType1
TypeArg2 -> RefType2
TypeArg3 -> RefType3

RefType1 ::= RefType '>'
RefType1 ::= ClassOrInterface '<' TypeArgList2
RefType2 ::= RefType '>>'
RefType2 ::= ClassOrInterface '<' TypeArgList3
RefType3 ::= RefType '>>>'
```

Figure 5.5 Production rules for parameterized types in the `ajc` grammar, working around incorrect tokenizations of parameterized types

deal with incorrect tokenizations, namely of parameterized types. For example, the parameterized type `List<List<List<String>>>` is tokenized by the `ajc` scanner as:

`List < List < List < String >>>`

where `>>>` is the unsigned right shift operator [3]. Because of this, the grammar cannot just define type arguments as a list of comma-separated types between '<' and '>', since in some cases the final > will not actually be a separate token.

This tokenization problem has to be dealt with in two places: in the `ajc` grammar and in the handwritten pattern parser. For the `ajc` grammar, Figure 5.5 shows the production rules for type arguments. Clearly, this is much more involved than it should be [4]. For the pattern parser, incorrect tokenizations of >> and >>> are fixed by splitting the tokens during parsing when the expected token is a single >. Figure 5.6 show the code for this. The `eat` method is used in the pattern parser to check if the next token is equal to a specified, `expected` token. If a shift operator is encountered, but a > is expected, then the token is split and the remainder of the token is stored in the variable `pendingRightArrows`, since the remainder is now the next token.

5.5.3 *Pseudo Keywords*

For compatibility with existing Java code, `ajc` does not reserve all the keywords introduced by AspectJ. Yet, the scanner of `ajc` *does* add keywords to the

[3] In C++ this is not allowed: a space is required between the angle brackets.
[4] This workaround is documented in the GJ specification [Bracha et al. 1998].

```
private void eat(String expected) {
  IToken next = nextToken();
  if(next.getString() != expectedValue) {
    if(expected.equals(">") && next.getString().startsWith(">")) {
      pendingRightArrows = substring from 1 of next;
      return;
    }
    throw parse error
  }
}

private IToken pendingRightArrows;
private IToken nextToken() {
  if(pendingRightArrows != null) {
    IToken ret = pendingRightArrows;
    pendingRightArrows = null;
    return ret;
  }
  else {
    return tokenSource.next();
  }
}
```

Figure 5.6 Splitting shift operators in the `ajc` pattern parser to work around incorrect tokenizations of parameterized types

lexical syntax of Java (`aspect`, `pointcut`, `privileged`, `before`, `after`, `around`, and `declare`), which usually implies that these keywords cannot be used as identifiers since the scanner will report these tokens as keywords. However, in its grammar, `ajc` introduces `JavaIdentifier`, a new non-terminal for identifiers, for which these keywords are explicitly allowed:

```
JavaIdentifier -> 'Identifier'
JavaIdentifier -> AjSimpleName

AjSimpleName -> 'around'
AjSimpleName -> AjSimpleNameNoAround
AjSimpleNameNoAround -> 'aspect' or 'privileged' or
    'pointcut' or 'before' or 'after' or 'declare'
```

This extended identifier replaces the original `Identifier`, which can no longer be one of the AspectJ keywords, at most places in the grammar. For example, the following productions allow the AspectJ keywords as the name of a class, method, local variable, and field.

```
ClassHeaderName1 ::= Modifiersopt 'class' JavaIdentifier
MethodHeaderName ::= Modifiersopt Type JavaIdentifier '('
VariableDeclaratorId ::= JavaIdentifier Dimsopt
```

However, the extended identifier is not allowed everywhere. In particular, it cannot be the first identifier of a type name, which means that it is not allowed as a simple type name, and cannot be the first identifier of a qualified type name, which could refer to a top-level package or an enclosing class. For example, the first import declaration is not allowed, but the second one is [5]:

[5]This is related to `ajc` bug 37069 at `https://bugs.eclipse.org/bugs/show_bug.cgi?id=37069`

```
import privileged.*;
import org.privileged.*;
```

If keywords would be allowed as simple type names, the grammar would no longer be LALR(1). The keywords as type names introduce shift-reduce and reduce-reduce conflicts. Hence, a qualified name is defined to be an `Identifier`, followed by one or more `JavaIdentifiers`:

```
ClassOrInterface ::= Name
SingleTypeImportDeclarationName ::= 'import' Name
Name -> SimpleName or QualifiedName
SimpleName -> 'Identifier'
QualifiedName ::= Name '.' JavaIdentifier
```

Pointcuts

The names of the primitive AspectJ pointcut designators, such as `get`, `set`, `call`, etc., are not declared as keywords. The scanner does not have any knowledge about pointcuts, so the names are parsed as identifiers, unless the pointcut designator was already a keyword, such as `if`. As we have seen earlier, the name `if` is still accidentally a reserved keyword, but the names of the other pointcut designators are not, so they can be used in pointcut expressions, for example in name patterns. However, a named pointcut with the same name as a primitive pointcut designator cannot be used (though surprisingly, it can be declared without warnings).

Around Advice Declarations

Around advice declarations introduce another complication. Whereas *after* and *before* advice declarations immediately start with the keywords `after` or `before`, *around* advice declarations start with a declaration of the return type. This introduces a shift-reduce conflict between an around advice declaration and a method declaration. For this reason, `ajc` does not allow methods named `around` in aspect declarations. Of course, it would not be acceptable to disallow the name `around` for all methods, including the ones in regular Java classes, so this restriction should only apply to aspect declarations (advice cannot occur in class declarations). Therefore, the `ajc` grammar needs to duplicate all the productions (19) from an aspect declaration down to a method declaration, where finally the name of a method is restricted to a `JavaIdNoAround`:

```
JavaIdNoAround -> 'Identifier'
JavaIdNoAround -> AjSimpleNameNoAround
MethodHeaderNameNoAround ::=
  Modifiersopt TypeParameters Type JavaIdNoAround '('
```

5.6 THE ABC SCANNER AND PARSER

The parser of `abc`[6] is based on Polyglot [Nystrom et al. 2003], which provides PPG, a parser generator for extensible grammars based on the LALR CUP parser generator. PPG acts as a front-end for CUP, by adding some extensibility and modularity features, which we will discuss later in Section 5.9.

[6] Our study is based on `abc` version 1.1.0, which supports `ajc` 1.2.1 with some minor differences

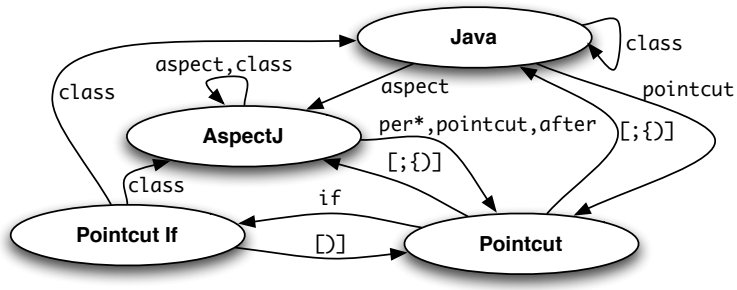

Figure 5.7 Lexical state transitions in the abc scanner

Polyglot's scanner for Java is implemented using the JFlex scanner generator. Polyglot does not feature an extensible scanner, so the abc compiler implements its own scanner for AspectJ, which takes an approach radically different from ajc. The abc scanner and parser can parse the entire source file in a single continuous parse. So, the Java, aspect, and pointcut language are defined in a single JFlex specification and CUP grammar. The abc scanner is designed to immediately produce the correct tokenization, so there is no need to fix incorrect tokenizations later. Also, the scanner does not interact with the parser.

5.6.1 *Managing Lexical State*

The abc scanner performs a rudimentary form of context-free parsing to recognize the global structure of the source file while scanning. The scanner keeps track of the current *state* (or *context*), by using a set of state transition rules that have been determined by a detailed analysis of the possible state switches in AspectJ. The lexical states and the transitions between them are illustrated in Figure 5.7. Some transitions have additional conditions, which we will explain later. Maintaining lexical state is not uncommon. It is widely used for scanning string literals and it is a standard feature of JFlex. Every lexical state has its own set of lexical rules, which means that a sequence of characters can be scanned as a different token in different states.

Pointcut Declarations

A simple example of such a state transition rule, is that a pointcut state is entered after the `pointcut` keyword and exited after a `";"` in pointcut context. For this example, the `pointcut` keyword and the semicolon indicates the start and end of a pointcut declaration, respectively. The exit of the pointcut state after a pointcut declaration is implemented in the flex specification by returning to the previous state (which is maintained on a stack) whenever the `";"` token is encountered in the pointcut state (`POINTCUT`):

```
<POINTCUT> {
  ";" {
```

```
      returnToPrevState();
      return op(sym.SEMICOLON);
    }
  }
```

For reasons of extensibility, keywords and their corresponding actions for *entering* lexical states are not specified in the flex specification, but are initialized from the Java code by means of a Java interface `LexerAction` whose instances can be registered with the scanner. `LexerActions` are always attached to keywords and can change the lexical state when the keyword has been scanned. For example, the following Java statement adds the keyword `pointcut`, which starts the pointcut declaration, to the scanner and specifies that the new lexical state after this keyword is pointcut.

```
lexer.addAspectJKeyword("pointcut",
  new LexerAction_c(new Integer(sym.POINTCUT),
    new Integer(lexer.pointcut_state())));
```

In this way, keywords are registered *per lexical state* in a `HashMap`. Initially, keywords are always scanned as identifiers and depending on the current lexical state, the identifier is turned into a keyword by a lexer action. As a side effect, the lexer action can modify the lexical state of the scanner. Figure 5.8 shows a fragment of the Java class `LexerAction_c` and the invocation of the lexer actions from the flex specification after an `Identifier` has been scanned. Note that keywords are automatically *reserved* in this way, since the identifier is always be turned in a keyword if there is a lexer action for it. Note that this design choice for reserved keywords is different from the pseudo keyword policy used by `ajc`.

If Pointcut Designator

The `pointcut` lexer action and the lexical rule for ; look rather concise, but unfortunately, most rules are more complex than this. For instance, the `if(..)` pointcut designator takes a Java expression as argument, which has the same lexical syntax as Java code in Java context, so the lexical state should be changed for the argument of the `if(..)`. Entering the lexical state is not very difficult: a lexer action for the `if` keyword can perform this state transition. The following Java statement adds the pointcut keyword `if` to the scanner and specifies that the new lexical state after this keyword is the special `POINTCUTIFEXPR` state:

```
lexer.addPointcutKeyword("if",
  new LexerAction_c(new Integer(sym.PC_IF),
    new Integer(lexer.pointcutifexpr_state())));
```

However, for recognizing the end of the `if(..)` pointcut designator, the scanner needs to find the closing parenthesis. Of course, a Java expression can contain parentheses as well. It would be incorrect to leave the special lexical state at the first closing parenthesis. Thus, the scanner needs to find the closing parenthesis that corresponds to the opening parenthesis after the `if`. For this purpose, the `abc` scanner maintains a variable `parenLevel` that is used to balance the parentheses. If a `")"` is encountered, the `parenLevel` is

```
<YYINITIAL,ASPECTJ,POINTCUTIFEXPR,POINTCUT> {
  {Identifier} {
    LexerAction la;
    switch(yystate()) {
      case YYINITIAL:
        la = javaKeywords.get(yytext());
        break;
      case ASPECTJ:
        la = aspectJKeywords.get(yytext());
        break;
      ...
    }

    if(la != null)
      return key(la.getToken(this));
    return id();
  }
}
```

```
class LexerAction_c implements LexerAction {
  public Integer token;
  public Integer nextState;

  public int getToken(AbcLexer lexer) {
    if(nextState != null)
      lexer.enterLexerState(nextState.intValue());
    return token.intValue();
  }
}
```

Figure 5.8 Lexer actions in the abc scanner

decremented and the new `parenLevel` is compared to the `parenLevel` of the `if` pointcut, for which the initial `parenLevel` has been saved in the entry on the `nestingStack`:

```
  <YYINITIAL,ASPECTJ,POINTCUTIFEXPR> {
    "(" {
      parenLevel++;
      return op(sym.LPAREN);
    }
    ")" {
      parenLevel--;
      if((yystate() == POINTCUTIFEXPR)
               && (parenLevel == nestingStack.peek().parenLevel))
        returnToPrevState();
      return op(sym.RPAREN);
    }
  }
```

Per-clause

There are more places where pointcuts can occur in an AspectJ program: aspect declarations optionally take a *per-clause*, which is used to control the instantiation of aspects. For example, declaring:

```
  aspect Foo perthis(pc) { ... }
```

entails that a new aspect instance of `Foo` is created for every `this` where the pointcut *pc* matches. Finding out the end of the pointcut of a per-clause is a bit more difficult than for normal pointcuts. The scanner again needs to find the matching closing parenthesis, but it also needs to know if it is actually scanning the pointcut of a per-clause or not. Instead of a new lexical state for per-clause pointcuts, the `abc` scanner uses a global boolean variable `inPerPointcut`. This variable is set to `true` by a lexer action for all per-clause keywords (`perthis`, `percflow`, etc.):

```
class PerClauseLexerAction_c extends LexerAction_c {
  ...
  public int getToken(AbcLexer lexer) {
    lexer.setInPerPointcut(true);
    return super.getToken(lexer);
  }
}
```

For a closing parenthesis in the pointcut lexical state, the scanner now needs to check if it is currently scanning a per-clause pointcut and if the closing parenthesis occurs at the same parenthesis level as the opening parenthesis that preceded the pointcut:

```
<POINTCUT> {
  ")" {
    parenLevel--;
    if(inPerPointcut &&
      parenLevel == nestingStack.peek().parenLevel) {
        returnToPrevState();
        inPerPointcut = false;
    }
    return op(sym.RPAREN);
  }}
```

Class Keyword

While the end of a lexical state is detected in the flex specification by a lexical rule for a token, the start of a context is declared in the lexer action of a keyword. In most cases, the start of a new lexical state is clearly indicated by a keyword. However, the `class` keyword does not unambiguously indicate the start of the Java lexical state for a class declaration, since it may also be used in class literals (e.g. `Foo.class`). To distinguish a class literal from a class declaration, the `abc` scanner maintains a special variable `lastTokenWasDot`. All tokens, except for the dot, set this variable to false. The rule for the `class` token can now determine whether it appears in a class literal or a class declaration and change the scanner state accordingly.

```
lexer.addGlobalKeyword("class",
  new LexerAction_c(new Integer(sym.CLASS)) {
    public int getToken(AbcLexer lexer) {
      if(!lexer.getLastTokenWasDot())
        lexer.enterLexerState(aspectj or java);
      return token.intValue();
    }
  });
```

It is interesting to observe the consequences for the scanner if a keyword no longer unambiguously indicates the next lexical state. In this case, the scanner needs to be updated for all tokens to maintain the `lastTokenWasDot` variable.

5.6.2 *Parser*

Thanks to the rudimentary context-free parsing in the scanner, the AspectJ grammar of `abc` is a clean modular extension of the basic Java grammar, implemented in PPG and based on the existing Polyglot grammar for Java. The grammar defines the entire AspectJ language, including pointcuts and name patterns, which is not the case in `ajc`.

Name Patterns

There is one interesting language construct for which some undesirable production rules have to be defined: name patterns. The grammar explicitly allows the reserved keywords of the pointcut lexical state as simple name pattern to allow name patterns such as `Foo.get`. Without explicitly allowing keywords, this would be forbidden, since `get` is a reserved keyword for pointcuts in `abc` and will therefore not be parsed as an identifier. The CUP production rules for this are:

```
simple_name_pattern ::=
  PC_MULT | IDENTIFIERPATTERN | IDENTIFIER
  | aspectj_reserved_identifier ;

aspectj_reserved_identifier ::=
  ASPECT | ... |  PC_GET | ... | PC_SET ... ;
```

This is somewhat unfortunate, because the keywords for pointcuts are hence defined in the grammar, as well as in the Java code, namely for adding lexer actions to the scanner. Extensions of AspectJ implemented in `abc` that introduce new pointcut keywords have to extend the `aspectj_reserved_identifier` production as well. Extensions may easily forget to do this and thereby reserve their keywords in name patterns. This extensibility issue will be discussed in more detail in Section 5.9.

Ideally, the `abc` scanner should enter a new lexical state for name patterns, since the lexical syntax of name patterns differs from pointcuts (i.e. the set of keywords is different). However, this will be more difficult to implement than the existing lexical states, since name patterns are not very explicitly delimited by certain tokens [7].

Parameterized Types

Although `abc` does not support AspectJ 5.0 and parameterized types, it is interesting to take a look at how the scanning problems for parameterized types would be solved in a similar setup of the scanner and parser. Currently, an extension of Polyglot for Java 5.0 is under development at McGill. In

[7] Indeed, very recently bug 72 has been created in the `abc` bugzilla, which proposes to introduce a lexer state for name patterns. See: `http://abc.comlab.ox.ac.uk/cgi-bin/bugzilla/show_bug.cgi?id=72`

```
reference_type_1 ::= reference_type GT
  | class_or_interface LT type_argument_list_2;
reference_type_2 ::= reference_type RSHIFT
  | class_or_interface LT type_argument_list_3;
reference_type_3 ::= reference_type:a URSHIFT;

wildcard   ::= QUESTION;
wildcard_1 ::= QUESTION GT;
wildcard_2 ::= QUESTION RSHIFT;
wildcard_3 ::= QUESTION URSHIFT;
```

Figure 5.9 Grammar production rules for parameterized types in Polyglot.

contrast to the approach of the abc compiler, the scanner of this extension does *not* always produce the correct tokenization for *regular Java*. Instead, the grammar works around the incorrect tokenization of parameterized types by encoding this in the definition of type arguments and reference types. To illustrate this workaround for incorrect tokenization, some production rules of this grammar are shown in Figure 5.9 (lots of details have been eluded). To resolve this issue a different lexical state should be used for types, since their lexical syntax is different from expressions. However, types will be very difficult to identify by a scanner in the input file, so this approach is rather unlikely to work.

Unfortunately, this grammar is now difficult to extend for reference types, since there are a large number of production rules involved, which encode the syntax of reference types in a rather tricky way.

5.7 SUMMARY AND DISCUSSION

We have discussed two approaches to parsing AspectJ. The ajc compiler uses a single scanner, but separate parsers (for 'regular' code and for pointcut expressions). The abc compiler uses a single parser with a stateful scanner. Based on our analysis we can make the following observations. Many rules on the syntax of AspectJ are only operationally defined in the implementation of the scanner and parser. As a consequence neither implementation provides a declarative formalization of the syntax of AspectJ, although the LALR grammar of abc [Hendren et al. 2004] is a step in the right direction. The ajc parser has undocumented implementation quirks because of the scanner implemented in and for plain Java. The abc parser improves over this by using a scanner with lexical states. The abc parser is also more predictable, but managing the lexical state in the parser is tricky and duplicates code and development effort. It is difficult to reason about the correctness and completeness of the context switching rules of the abc scanner. For example, the use of the global variable inPerPointcut happens to work correctly in case an anonymous class is used with aspect members in a per-pointcut, but a slight change or extension of the language may render this implementation invalid. Choices for introducing lexical states are guided by the complexity of determining this lexical state in the scanner. For example, a separate lexical state

for name patterns might be more appropriate. In conclusion, although the implementation techniques used in the parsers of `ajc` and `abc` are effective for parsing AspectJ, their implementations have several drawbacks.

5.8 A DECLARATIVE SYNTAX DEFINITION FOR ASPECTJ

In the previous sections, we have presented a range of implementation issues in parsing AspectJ, and the solutions for these in the two major AspectJ compilers, i.e. `ajc` and `abc`. As a consequence of these issues, the syntax of the language that is supported by these compilers is not clearly defined. We conclude that the grammar formalisms and parsing techniques that are used are not suitable for the specification of the AspectJ language. A complete and declarative definition of the syntax of the AspectJ language is lacking.

In this section, we present a definition of the syntax of AspectJ that is declarative, modular, and extensible. Our AspectJ syntax definition is based on the syntax definition formalism SDF and its implementation with scannerless generalized LR parsing. Thanks to these foundations, the definition elegantly deals with the extension and embedding of the Java language, the problems of context-sensitive lexical syntax, and the different keyword policies of the `ajc` and `abc` compilers. Indeed, the modularity of SDF allows us to define three variants of the AspectJ language:

- AJF, which is the most liberal definition, where only real ambiguities are resolved, for example by reserving keywords at very specific locations.
- AJC, which adds restrictions to the language to be more compatible with the official AspectJ compiler. The additional restrictions are mostly related to shift-reduce problems in the LALR parser of `ajc`.
- ABC, which reserves keywords in a context-sensitive way, thus defining the language supported by the `abc` compiler.

The AspectJ syntax definition modularly extends our syntax definition for Java 5 [8]. Also, the AJF, AJC, and ABC variants are all modular extensions of the basic AspectJ definition. Moreover, in Section 5.9 we will show that our syntax definition can easily be extended with new aspect features. In Section 5.10 we present benchmark results, which show that these techniques yield a parser that performs linear in the size of the input with an acceptable constant factor, at least for specification, research and prototyping purposes.

The core observation underlying the syntax definition is that AspectJ is a combination of languages, namely Java, aspects, and pointcuts. From this viewpoint, this work applies and extends previous work on combining languages for the purpose of domain-specific language embedding [Bravenboer & Visser 2004 (Chapter 2)] and metaprogramming with concrete object syntax [Bravenboer et al. 2005 (Chapter 3)] (see Section 5.11.1).

[8] Available at `http://java.syntax-definition.org`

5.8.1 *Integrating Lexical and Context-Free Syntax*

SDF integrates the definition of lexical and context-free syntax in a single formalism, thus supporting the *complete* description of the syntax of a language in a single definition. In this way, the lexical syntax of AspectJ can be integrated in the context-free syntax of AspectJ, which automatically leads to *context-sensitive* lexical syntax. Parsing of languages defined in SDF is implemented by the scannerless generalized LR parser SGLR [Visser 1997a], which operates on individual characters instead of tokens. Thus, recognizing the lexical constructs in a source file is actually the same thing as parsing. This solves most of the issues in parsing AspectJ.

Lexical syntax can be disambiguated in a declarative, explicit way, as opposed to the implicit, built-in heuristics of lexical analysis tools, such as a longest-match policy and a preference for keywords. Without explicit specification, keywords are *not* reserved and, for example, are perfectly valid as identifiers. Instead, keywords can be reserved explicitly by defining *reject productions*.

Java

Figure 5.10 illustrates the basic ideas of SDF with sample modules and productions from the Java syntax definition. Of course, the real syntax definition is much larger and spread over more modules. Note that the arguments of an SDF production are at the left and the resulting symbol is at the right, so an SDF production $s_1 \ldots s_n$ -> s_0 defines that an element of nonterminal s_0 can be produced by concatenating elements from nonterminals $s_1 \ldots s_n$, in that order. The modules of Figure 5.10 illustrate that modules have names [14] and can import other modules [15]. The module `Java` defines the composition of compilation units [16] from package declarations, import declarations, and type declarations. Note the use of optional (`?`) and iterated (`*`,`+`) nonterminals. The module `Expressions` defines expression names [17] (local variables, fields, etc), addition of expressions [18], which is declared to be `left` associative, and method invocation [19]. The production rule for method invocations uses `{`*s lit*`}*`, which is concise notation for a list of *s* separated by *lit*. The module `Identifiers` shows how *lexical syntax* is defined in the same syntax definition as the *context-free syntax*. To define lexical nonterminals such as identifiers SDF provides character classes to indicate sets of characters [20]. The `Identifiers` module also defines a longest-match policy for identifiers, by declaring that identifiers cannot directly be followed by one of the identifier characters [21]. Another difference with respect to other formalisms is that there may be multiple productions for the same nonterminal. This naturally leads to *modular* syntax definitions in which syntax can be composed by importing modules.

Aspects

Similar to the syntax definition of Java, SDF can be used to define modules for the languages of aspects, pointcut expressions, and patterns. Figure 5.11 presents a few productions for aspect declarations in AspectJ. The first two

```
module Java [14]
imports Statements Expressions Identifiers [15]
exports
   context-free syntax
     PackageDec? ImportDec* TypeDec+ -> CompilationUnit [16]
```

```
module Statements
exports
  context-free syntax
    "for" "(" FormalParam ":" Expr ")" Stm -> Stm
    "while" "(" Expr ")" Stm               -> Stm
```

```
module Expressions
exports
  context-free syntax
    ExprName                        -> Expr              [17]
    Expr "+" Expr                   -> Expr {left}       [18]
    MethodSpec "(" {Expr ","}* ")" -> Expr               [19]
    MethodName                      -> MethodSpec
    Expr "." TypeArgs? Id           -> MethodSpec
```

```
module Identifiers
exports
  lexical syntax
    [A-Za-z\_\$][A-Za-z0-9\_\$]* -> Id                   [20]

  lexical restrictions
    Id -/- [a-zA-Z0-9\_\$]                               [21]
```

Figure 5.10 Fragment of syntax definition for Java

productions define aspect declarations [22] and aspect declaration headers [23]. Both productions use nonterminals from Java, for example `Id`, `TypeParams` (generics), and `Interfaces`. The aspect header may have a per-clause, which can be used to control the instantiation scheme of an aspect. For instance, a `perthis` clause [24] specifies that one aspect instance is created for each currently executing object (`this`) in the join points matched by the pointcut expression given as a parameter. The per-clause `pertypewithin` [25], which has been added to the language in AspectJ 5, is used to create a new aspect instance for each type that matches the given type pattern. This is the only per-clause that does not take a pointcut expression as an argument.

Advice declarations [26] are mainly based on an advice specifier and a pointcut expression, where an advice specifier can be a before [27], after [28], or around [29] advice. Note that most of the productions again refer to Java constructs, for example `ResultType` and `Param` (an abbreviation of `FormalParam`).

Pointcuts

The AspectJ pointcut language is a language for concisely describing a set of join points. Pointcut expressions consist of applications of pointcut designators, which can be primitive or user-defined. Also, pointcut expressions

```
module AspectDeclaration
exports
  context-free syntax
    AspectDecHead AspectBody -> AspectDec    22
    AspectMod* "aspect" Id TypeParams? Super?
          Interfaces? PerClause? -> AspectDecHead    23

    "perthis"       "(" PointcutExpr ")" -> PerClause    24
    "pertypewithin" "(" TypePattern ")"  -> PerClause    25

    AdviceMod* AdviceSpec Throws? ":" PointcutExpr
                                    MethodBody -> AdviceDec    26

    "before" "(" {Param ","}* ")"              -> AdviceSpec    27
    "after"  "(" {Param ","}* ")" ExitStatus? -> AdviceSpec    28
    ResultType "around" "(" {Param ","}* ")"  -> AdviceSpec    29
    "returning" "(" Param ")" -> ExitStatus    30
```

Figure 5.11 Fragment of syntax definition for aspects and advice

```
module PointcutExpression
exports
  context-free syntax
    "call"   "(" MethodConstrPattern ")"    -> PointcutExpr  31
    "get"    "(" FieldPattern ")"           -> PointcutExpr  32
    "this"   "(" TypeIdStar ")"             -> PointcutExpr  33
    "cflow"  "(" PointcutExpr ")"           -> PointcutExpr  34
    "if"     "(" Expr ")"                   -> PointcutExpr  35
    PointcutName "(" {TypeIdStar ","}* ")" -> PointcutExpr  36
    Id -> PointcutName
```

Figure 5.12 Fragment of syntax definition for AspectJ pointcut expressions.

can be composed using boolean operators. Figure 5.12 shows some of the primitive pointcuts of AspectJ. The `call` 31 and `get` 32 pointcut designators take patterns of methods, constructors, or fields as arguments. The `this` 33 pointcut designator cannot be used with arbitrary type patterns. Instead, the argument must be a `Type`, an `Id` or a wildcard. The `if` 35 pointcut designator, which we have discussed before, takes a boolean Java expression as an argument. Finally, Figure 5.12 defines the syntax for user-defined pointcuts 36 in pointcut expressions, which have been declared somewhere in the program using a pointcut declaration.

Patterns

The AspectJ pattern language plays an important role: as we have already seen, most of the pointcut designators operate on patterns. Figure 5.13 shows some productions for the syntax of the pattern language. *Name patterns* are used to pick out names in a program. A name pattern is a composition of identifier patterns 44, which are used for matching identifiers (i.e. names without a dot) by adding a `*` wildcard to the set of identifier characters. The `..` wildcard 37 can be used to include names from inner types, subpackages, etc. Almost every pointcut uses *type patterns*, which are used for selecting types.

```
module Pattern
exports
  context-free syntax
    IdPattern                  -> NamePattern
    NamePattern "."  IdPattern -> NamePattern
    NamePattern ".." IdPattern -> NamePattern          37

    PrimType           -> TypePattern                  38
    TypeDecSpecPattern -> TypePattern
    TypeDecSpecPattern TypeParamsPattern -> TypePattern 39
    NamePattern        -> TypeDecSpecPattern           40
    NamePattern "+"    -> TypeDecSpecPattern           41

    FieldModPattern TypePatttern ClassMemberNamePattern
                                                  -> FieldPatttern 42
    MethodModPattern TypePattern ClassMemberNamePattern
        "(" {FormalPattern ","}* ")" ThrowsPattern? -> MethodPattern 43

lexical syntax
  [a-zA-Z\_\$\*][a-zA-Z0-9\_\$\*]* -> IdPattern  44
```

Figure 5.13 Fragment of syntax definition for AspectJ patterns

Any name pattern is a type pattern [40], but type patterns can also be used to match subtypes [41], primitive types [38], parameterized types [39], etc.

Method [43] and *field patterns* [42] combine name patterns, type patterns, modifier patterns, throw patterns and patterns for formal parameter into complete signature patterns that are used to match methods and fields by their signatures.

5.8.2 *Composing AspectJ*

We have now illustrated how the syntax of the sublanguages of AspectJ (Java, aspects, pointcuts, and patterns) can be defined as separate SDF modules. Next, we need to compose these modules into a syntax definition for AspectJ itself. In SDF, we can combine two syntax definitions by creating a new module that imports the main modules of the languages that need to be combined. The ease with which syntax definitions can be composed, is due to the two main features of the underlying parser: *scannerless parsing* and the use of the *generalized LR* algorithm.

First, in a setting with a separate scanner such a combination would cause conflicts as has extensively been discussed in Section 5.4. However, in the scannerless SDF setting this does not pose a problem. Since lexical analysis is integrated with parsing, context-sensitive lexical analysis comes for free. For example, when parsing `1+1` as a Java expression the `+` will be seen as an addition operator [18], but when parsing `Foo+` in the context of a pointcut expression, then the `+` will be interpreted as a subtype pattern [41].

Second, if LL, LR, or LALR grammars are used, then the combination of one or more languages is not guaranteed to be in the same subset, since these subsets of the context-free languages are not closed under composition. Indeed, if we combine method declarations from Java and advice declarations

```
module AspectJ [45]
imports
  Java AspectDeclaration PointcutExpression Pattern [46]
exports
  context-free syntax
    AspectDec     -> TypeDec            [47]
    ClassBodyDec -> AspectBodyDec       [48]
    AspectDec     -> ClassMemberDec     [49]
    PointcutDec   -> ClassMemberDec     [50]
```

Figure 5.14 SDF module combining Java, pointcut, and aspect declarations.

from aspects, then shift-reduce conflicts pop up since this combination is no longer LALR, as has been discussed in Section 5.5.3. Since SDF is implemented using generalized LR parsing, SDF supports the full class of context-free grammars, which *is* closed under composition. Hence, new combinations of languages will stay inside the same class of context-free grammars.

Nevertheless, in some cases there will be ambiguities in the new combination of languages where there are actually two or more possible derivations for the same input. These ambiguities can be solved in a declarative way using one of the SDF disambiguation filters [van den Brand et al. 2002], such as reject, priorities, prefer, and associativity. Section 5.9 presents examples of this in AspectJ extensions. However, this is not the case for AspectJ. For example, the around advice problem is not a real ambiguity: the syntax of around advice and method declarations are similar for the first few arguments, but the colon and the pointcut expression distinguishes the around advice syntactically from method declarations.

AspectJ

Figure 5.14 illustrates how the languages can be combined by importing [46] the modules of Java, aspects, pointcuts, and patterns [9]. In this way, most of the integration happens automatically: the productions for pointcut expressions already refer to patterns and aspect declarations already refer to pointcut expressions and patterns. By importing all modules, the symbols and productions of these modules will be combined, and as a result the pointcut expressions will automatically be available to the aspect declarations.

The integration of the languages can be extended and refined by adding more productions that connect the different sublanguages to each other. For instance, aspect declarations (`AspectDec`) are Java type declarations, since they can be used at the top-level of a source file [47] (see also the production rule for compilation units [16]). Furthermore, aspect declarations [49] and pointcut declarations [50] can occur *inside* a class, i.e. as members of a Java class declaration.

Just as aspects and pointcuts can be defined in regular Java code, the declarations of aspects can contain Java members such as constructors, initializers, fields, and method declarations. Thus, Java class body declarations

[9]The actual composition in the full definition is somewhat different, to make the definition more customizable. We will discuss this later.

(`ClassBodyDec`, i.e. elements of a Java class body) are allowed as aspect body declarations [48].

5.8.3 *Disambiguation and Restrictions*

We have not yet defined any reserved keywords or other restrictions for the syntax that we have presented. Next, we explain how the syntax definition can be extended in a *modular* way to impose additional restrictions on the language, such as different reserved keyword policies and requirements for being compatible with the language accepted by an LALR grammar. First, we discuss how keywords can be reserved in SDF. Next, we discuss the real ambiguities of the language that we have presented so far. The resulting syntax definition, which is the most liberal AspectJ syntax definition without ambiguities, is called AJF. After that, we extend the restriction to achieve the AJC and ABC variants, which are designed to be compatible with the AspectJ language as supported by the `ajc` and `abc` compilers, respectively.

Reserving Keywords

Scannerless parsing does not *require* a syntax definition to reserve keywords. Depending on the context, the same token can for example be interpreted as a keyword or as an identifier. However, in some cases a keyword is inherently ambiguous if it is not reserved. For example, the Java expressions `this` and `null` would be ambiguous with the identifiers `this` and `null` if they would not be reserved. In SDF reserved keywords are defined using *reject productions* [Visser 1997a], which are productions annotated with the `reject` keyword. The following two SDF productions illustrate this mechanism:

```
"abstract" | "assert" | ... | "while" -> Keyword
Keyword -> Id {reject}
```

The first production defines keywords and the second rejects these keywords as identifiers. Reject productions employ the capability of generalized LR parsers to produce all possible derivations. In case of a keyword, there will be two possible derivations: one using the real production for identifiers and one using the reject production. If the reject production is applicable, then all possible parses that produce the same nonterminal (in this case `Id`) are eliminated. In this way, the parse that uses the production for the real identifier is disallowed. Thus, in SDF reserved keywords are defined *per nonterminal*: in the example above, the keywords are *only* reserved for the `Id` nonterminal. If other identifier-like nonterminals would exist in Java (which is not the case), then keywords would not be reserved for that nonterminal. Because there is just a single identifier nonterminal for regular Java, this feature does not add much over a mechanism for global keywords, but the feature is most useful if languages are being combined: it can be used for defining context-sensitive keywords.

AJF

One of the few ambiguities in the syntax definition are the applications of user-defined [36] and primitive pointcut designators. For example, the pointcut expression `this(Foo)` can be parsed as the primitive pointcut `this`, but it can also parsed as a user-defined pointcut with the same name. To resolve this ambiguity, AJF rejects the names of primitive pointcuts as the name of a user-defined pointcut, which is similar to the behaviour of `ajc` and `abc`. To make the names of primitive pointcuts available to extensions and the other variants of AspectJ, we introduce a new nonterminal: `PrimPointcutName`. These names are rejected as the name of a used-defined pointcut.

```
"adviceexecution" | "args" | "call" | ...
    | "within" | "withincode" -> PrimPointcutName
PrimPointcutName -> PointcutName {reject}
```

Another ambiguity that needs to be resolved by reserving keywords occurs in type patterns. Type patterns are composed of name and identifier patterns, but we have not imposed any restrictions on these name patterns, which implies that a name pattern can just as well be one of the built-in types `int`, `float`, `void`, etc. We do not want to reject these types as identifier patterns in general, since there is actually no ambiguity there. To resolve this ambiguity more precisely, we can disallow keywords only for the name patterns that are used as type patterns, i.e. `TypeDecSpecPatterns` [40].

```
Keyword -> TypeDecSpecPattern {reject}
```

The final ambiguity is a bit more surprising. The `ajc` compiler does not reserve keywords in patterns, not even the regular Java keywords (except for the bug with the `if` pseudo token). For example, the method pattern `* try(String)` is accepted by `ajc`. Of course, this is not very useful since there can never be a method with this name, but for now we follow this decision. As a result of this, the identifier pattern `new` is allowed for the name of a method in a method pattern. Surprisingly, the constructor pattern `*Handler+.new()` can now also be parsed as a method pattern by splitting the `*Handler` identifier pattern after any of its characters. The part before the split then serves as a type pattern for the return type of the method. For example, one of the results of parsing are the method patterns `* Handler+.new()` and `*H andler+.new()`. The reason for this is that SDF does not by default apply a longest-match policy. Of course, this split is not desirable, so to disallow this, we define a longest-match policy *specifically* for identifier patterns using a follow restriction, which forbids derivations where an identifier pattern is followed by a character that can occur in a pattern.

```
IdPattern -/- [a-zA-Z0-9\_\$\*]
```

AJC *Compatibility*

In Section 5.5.3 we have discussed the pseudo keyword policy of `ajc` in detail. Basically, the pseudo keywords of AspectJ are only reserved for a few specific language constructs. This can concisely be expressed using reject productions,

which allow the definition of reserved keywords *per nonterminal*. Similar to the `PrimPointcutName` we introduced earlier, a new nonterminal for pseudo keywords can be used. For all the language constructs that cannot be pseudo keywords, a reject production is defined. For example:

```
"aspect" | "pointcut" | "privileged" | "before"
    | "after" | "around" | "declare" -> PseudoKeyword
PseudoKeyword -> TypeName          {reject}
PseudoKeyword -> PackageOrTypeName {reject}
```

The first production handles the case where a typename is a single identifier (e.g. `aspect`). The second case rejects pseudo keywords as the first identifier of the qualifier of a typename (i.e. a package- or typename), which corresponds to the behavior of `ajc`, where pseudo keywords are not allowed as the first identifier of a typename. Finally, to be more compatible with `ajc`, AJC could produce parse errors for incorrect floating-point literals in name patterns by defining the syntax of incorrect floating-point literals and the name patterns that contain them. These patterns can then be rejected as name patterns. If this behaviour were required, then this might be useful, but for now we leave this as an 'incompatibility'.

ABC *Compatibility*

While extending the syntax definition for compatibility with `ajc` was relatively easy, extending the definition (as we have presented it until now) to become compatible with `abc` is substantially more difficult, if undertaken without the appropriate solutions. First, we discuss how a relatively easy restriction of `abc` can be enforced. This leads to the explanation why other restrictions are impossible to solve concisely in the current setup. For this, and for the definition of AspectJ extensions, we present a novel method of combining languages using *grammar mixins*. Grammar mixins then arise as the key mechanism for composing the languages involved in AspectJ. After discussing grammar mixins, we return to the ABC compatibility.

KEYWORDS AND NAME PATTERNS In Section 5.6 we have discussed that the `abc` compiler reserves a different set of keywords per lexical state. For example, in the lexical state of a pointcut, `abc` reserves all the names of primitive pointcut designators. To support these keywords (such as the rather common `get` and `set`) in identifier patterns, they are explicitly allowed by the grammar of `abc` (Section 5.6.2). In SDF, this is not an issue: keywords are reserved *per nonterminal*, so keywords that have been reserved for identifiers are still allowed as identifier patterns. As opposed to `ajc`, `abc` does *not* allow regular Java keywords as identifier patterns, so the previous example of the method pattern `* try(String)` results in a syntax error. In our ABC compatible variant, this is handled by rejecting plain Java keywords as identifier patterns:

```
Keyword -> IdPattern {reject}
```

However, it is not obvious how the context-sensitive keywords of `abc` could be defined. For example, consider the following candidate for making primitive pointcut names keywords:

```
PrimPointcutName -> Keyword
```

Unfortunately, adding this production reserves keywords in every context, not just in pointcuts. The previous reject production for `IdPattern` illustrates why this is the case: we only have a single keyword nonterminal and in this way we cannot have context-specific sets of keywords. Moreover, we have just a single identifier nonterminal (`Id`), but an identifier can occur in every context, and for every context we need to reserve a different set of keywords. Since we cannot refer to an identifier in a specific context, it is impossible to define reserved keywords for it. Grammar mixins are a solution for this, but are more generally useful than just for defining reserved keywords.

5.8.4 *Grammar Mixins*

In the context of object-oriented programming, mixins are abstract subclasses that can be applied to different superclasses (i.e. are parameterized in their superclass) and in this way can form a family of related classes [Bracha & Cook 1990]. In the context of grammars, grammar mixins are syntax definitions that are parameterized with the context in which they should be used. The key observation that leads to the use of mixins for defining AspectJ is that the language uses multiple instances of Java, which are mixed with the new language constructs of AspectJ. For example, a Java expression in the context of an `if` pointcut is different from a Java expression in an advice declaration or in a regular Java class. Similarly, an identifier in the context of a pointcut is different from an identifier in an aspect body declaration. Therefore, it should be possible to handle them as separate units, which would make it possible to customize them separately.

Therefore, the Java language should be reusable in the definition of a new language, where the Java syntax effectively becomes part of the new syntax definition, i.e. if syntax definition $\mathcal{A}_1$ imports $\mathcal{B}$ and $\mathcal{C}$ using mixin composition, then the syntax of $\mathcal{B}$ and $\mathcal{C}$ should effectively become part of $\mathcal{A}_1$. A different language $\mathcal{A}_2$ should be able to compose itself with $\mathcal{B}$ or $\mathcal{C}$ and modify this new composition without affecting the other combination of $\mathcal{A}_1$, $\mathcal{B}$, and $\mathcal{C}$.

Grammar mixins provide a more flexible way of composing languages compared to the plain import mechanisms of SDF that we have been using until now. Using grammar mixins, Java can be mixed with pointcuts, name patterns, and aspects and each of these combinations is again a unit for composition. Also, it is possible to extend, customize, or restrict the Java language only for some specific combination. In particular, SDF grammar mixins flourish because the syntax definitions that are subject to mixin compositions are complete: the lexical as well as the context-free syntax is being composed and can *both* be customized for a specific composition. In the next section we will show how grammar mixins can be used to their full potential to combine AspectJ language extensions by unifying mixin compositions.

SDF Implementation For the implementation of grammar mixins we make use of a combination of existing SDF features whose applicability to syntax

```
module JavaMix[Ctx]   51
imports Java          52
  [ CompilationUnit => CompilationUnit[[Ctx]]   53
    TypeDec         => TypeDec[[Ctx]]
    ...
    FieldAccess     => FieldAccess[[Ctx]]
    MethodSpec      => MethodSpec[[Ctx]]
    Expr            => Expr[[Ctx]]  ]
```

Figure 5.15 SDF grammar mixin for Java.

```
module AspectJ[JavaCtx AspectCtx PointcutCtx PatternCtx]
imports
  JavaMix[JavaCtx]                              54
  JavaMix[AspectCtx]
  JavaMix[PointcutCtx]
  JavaMix[PatternCtx]
  aspect/Declaration[AspectCtx JavaCtx]         55
  pattern/Main[PatternCtx]                      56
  pointcut/Expression[PointcutCtx JavaCtx]      57
```

Figure 5.16 Main module of grammar mixin-based AspectJ

definition had not been fully explored previously: parameterized modules and parameterized symbols. Figure 5.15 shows the SDF implementation of the mixin module for Java. An SDF grammar mixin is an SDF module that has a formal parameter 51 that identifies a particular mixin composition. By convention this parameter is called `Ctx` (for context) and the module name has the suffix `Mix`. This grammar mixin module imports the real syntax definition 52 and applies a renaming 53 to all the nonterminals of the grammar, which places these nonterminals in the given `Ctx` by using a parameterized nonterminal. The list of renamings covers all the nonterminals of the language, which can be a very long list that is tedious to maintain. Therefore, we provide a tool `gen-sdf-mix` that generates a grammar mixin module given an SDF syntax definition. The grammar mixin is never modified by hand, so it can be regenerated automatically.

All grammar mixins that are imported using the same symbol for `Ctx` are subjected to mixin composition. In a way, the import statement of SDF and `Ctx` symbol are the mixin composition operators of grammar mixins. For grammar mixins, composition means that the grammars of the syntax definitions involved in a composition are fully automatically combined, based on the normal SDF grammar composition semantics (which are also applied to plain imports).

5.8.5 *AspectJ in the Mix*

Now we have revealed the actual design of the syntax definition, we need to revise the presentation of the AspectJ syntax. Figure 5.16 shows the imports of the main module of the syntax definition. The AJF, AJC, and ABC variants im-

```
AspectDec                 -> TypeDec[[JavaCtx]]                (see 47)
ClassBodyDec[[AspectCtx]] -> AspectBodyDec                     (see 48)
AspectDec                 -> ClassMemberDec[[JavaCtx]]         (see 49)
PointcutDec               -> ClassMemberDec[[JavaCtx]]         (see 50)

"before" "(" {Param[[AspectCtx]] ","}* ")" -> AdviceSpec       (see 27)
"if" "(" Expr[[JavaCtx]] ")" -> PointcutExpr                   (see 35)
 PrimType[[PatternCtx]]      -> TypePattern                    (see 38)
```

Figure 5.17 AspectJ productions updated to grammar mixins. The numbers refer to the productions mentioned earlier.

port this module and the variant specific modules. The `AspectJ` module itself has four contexts parameters, to make the mixin composition configurable for AspectJ extensions. `AspectJ` imports the grammar mixin `JavaMix` four times, once for every context. This makes all the nonterminals of Java available to AspectJ in these four contexts. The choice of the four contexts is somewhat arbitrary. For example, it might be a good idea to introduce an additional context for advice. Fortunately, this is very easy to do by just importing another instance of the Java grammar mixin with a symbol for that context. Our syntax definition has one context more than the `abc` scanner has lexical states: `abc` does not place patterns in a separate context.

Next, the modules for the sublanguages are imported, passing the required contexts as parameters to the modules. For example, pointcut expressions 57 need to know their own context, but also the context of regular Java expressions.

The imports of `JavaMix` and the sublanguage modules automatically compose all mixin compositions, but we still need to make some interactions explicit, like we did earlier in Figure 5.14. However, this time the productions also connect nonterminals from different contexts (mixin compositions). Figure 5.17 shows some of the production rules that we have discussed earlier, but this time using the context parameters. For example, aspect declarations are type declarations in the `JavaCtx` 47, but all the arguments of the aspect declaration will be in the context of aspects, so an aspect declaration changes the context from `JavaCtx` to `AspectCtx` in this case. The second production 48 defines that regular Java class body declarations from the aspect context can be used as aspect body declarations. The productions for aspect 49 and pointcut declarations 50 make these constructs available as class members in the regular Java context. Advice specifiers 27 use Java's formal parameters from the aspect context. The `if` pointcut expression takes an expression from the regular Java context as an argument. For Abc compatibility, we will later define reserved keywords *per context*. By using the expression from the Java context, aspect and pointcut-specific keywords will be allowed in this Java expression. Finally, the type pattern for primitive types 38 now uses a primitive type from the pattern context.

Note that the choice of the context of a symbol is completely up to the language designer: for every production argument we can choose the most

appropriate context. The choice of the context switches (lexical state transitions) is not influenced by the complexity of recognizing the context during lexical analysis. In the next section we show that this enables language designers to improve their language designs.

5.8.6 Abc *Compatibility Revised*

Thanks to the grammar mixins, we can now declare a different set of reserved keywords for each context. The AspectJ grammar now has four nonterminals for identifiers: `Id[[JavaCtx]]`, `Id[[AspectCtx]]`, `Id[[PointcutCtx]]`, and `Id[[PatternCtx]]`. Similarly, there are four nonterminals for keywords. Thus, the syntax definition can now reject a different set of reserved keywords for each specific context. The reject production is in fact already defined in the Java modules imported by the AspectJ definition, so we only need to extend the existing set of keywords. For the Java context, `abc` introduces three new keywords:

```
"privileged" | "aspect" | "pointcut" -> Keyword[[JavaCtx]]
```

For the aspect context, `abc` introduces a series of new keywords. Also, every keyword from the Java context is a keyword in aspect context.

```
"after" | ... |"proceed" -> Keyword[[AspectCtx]]
Keyword[[JavaCtx]] -> Keyword[[AspectCtx]]
```

However, `proceed` is now a reserved keyword in aspect declarations, so it is no longer allowed as the name of a method invocation, which now rejects the special `proceed` call for invoking the original operation in an `around` advice. To reintroduce the `proceed` call, we need to allow it explicitly as a method specifier in the aspect context (note that an advice context would be useful here, though that would not be compatible with `abc`, which is the whole point of this exercise).

```
"proceed" -> MethodSpec[[AspectCtx]]
```

In the context of pointcuts, `abc` reserves the Java keywords, primitive pointcut names, and some additional keywords from the context of aspects.

```
Keyword[[JavaCtx]]          -> Keyword[[PointcutCtx]]
PrimPointcutName            -> Keyword[[PointcutCtx]]
"error" | ... | "warning" -> Keyword[[PointcutCtx]]
```

Finally, we still need to define keywords for the context of patterns, since our syntax definition uses a separate context for that. In `abc`, these two states are merged, so defining pattern keywords is easy:

```
Keyword[[PointcutCtx]] -> Keyword[[PatternCtx]]
Keyword[[PatternCtx]]  -> IdPattern {reject}
```

We have now defined the keyword policy of `abc` in a declarative way as a modular extension of the basic syntax definition.

In the last few years, there has been a lot of research on extensions of AspectJ. For experimenting with aspect-oriented language features, an *extensible compiler* for AspectJ is most useful. One of the goals of the `abc` project is to facilitate this research by providing such an extensible compiler. The previous sections have highlighted a few challenges for the definition of the syntax of AspectJ and the implementation of an AspectJ parser. The result of this complexity is that the parsers of `ajc` and `abc` are more complex than usual, since the requirements imposed on the parser by the language do not match the conventional parsing techniques too well.

This section demonstrates these limitations through several existing extensions and their issues. We compare the implementation of the *syntax* of the extensions in `abc` to the definition of the syntax in SDF, based on the syntax definition for AspectJ that we presented in the previous section. We would like to emphasize that this discussion is all about the *syntax* of the extensions, and not about the other compiler phases. Our modular and declarative approach for the definition of the syntax of AspectJ does not suddenly make the *complete* implementation of AspectJ extensions trivial, since a lot of work is going on in later compiler phases.

5.9.1 *Issues in Extensibility*

The `abc` compiler is based on Polyglot [Nystrom et al. 2003], which provides PPG, a parser generator for extensible grammars based on CUP, a LALR parser generator. The extensibility features of PPG are based on manipulation of grammars, with features such as `drop` a symbol, `override` productions of a symbol, and `extend` the productions of a symbol. This way of extending a grammar works in practice for most of the language extensions that have been implemented for `abc` until now. Unfortunately this is not a truly modular mechanism, since LALR grammars do not compose, which means that the user of PPG has to make sure that the composed grammar stays in the LALR subclass of context-free grammars. For example, we have discussed the problem of around advice and method declarations with the name `around`. The `abc` compiler overcomes some of these issues by reserving keywords.

PPG does not feature an extensible scanner, so the `abc` compiler implements its own, stateful scanner as we have discussed in detail. This works fine for the basic AspectJ language, but it is inherently not modular. The rules for switching from context are based on knowledge of the entire language that is being scanned, which breaks down if the language is extended in an unexpected way. The `abc` scanner allows extensions to add keywords to specific states of the scanner. In this way, it is relatively easy to add keywords, but it is difficult to add operators and it is much more difficult to add new scanner states. For example, suppose that AspectJ did not define an `if(...)` pointcut. It would have been non-trivial to extend the scanner to handle this pointcut, since it requires the introduction of a new lexical state that affects several aspects of

```
module HelloWorld[JavaCtx AspectCtx PointcutCtx]
exports
  context-free syntax
    "cast" "(" TypePattern ")" -> PointcutExpr 58

    "global"":" ClassNamePattern ":" PointcutExpr ";" -> PointcutDec 59

    "cflowlevel" "(" IntLiteral[[JavaCtx]] "," PointcutExpr ")"
                                                    -> PointcutExpr 60
  lexical syntax
    "cast"       -> Keyword[[PointcutCtx]]
    "cflowlevel" -> Keyword[[PointcutCtx]]
    "global"     -> Keyword[[JavaCtx]]
    "global"     -> Keyword[[AspectCtx]]
```

Figure 5.18 Syntax of some abc extensions implemented in SDF

the scanner. In these situations, the scanner has to be copied and modified, which is undesirable for maintenance and composition of extensions.

The modular syntax definition we have presented solves many of these issues, since the definition itself can be extended in a modular way as well. Context or lexical state management is not based on rudimentary context-free parsing in the scanner, but fully integrated in the parser by the use of scannerless parsing. Moreover, contexts can be unified by mixin composition and ambiguities can be resolved in a modular way.

5.9.2 *Simple Extensions*

First, we discuss some small AspectJ extensions that are part of the EAJ (Extended AspectJ) extension of abc. The SDF implementation of the extensions is shown in Figure 5.18. Similar to the way Java is extended, the AspectJ syntax definition can be extended by creating a new module that imports AspectJ and adds new constructs.

Cast and Global Pointcuts

The cast pointcut designator 58 can be used to select points in the program where an implicit or explicit cast is performed. This is a very simple pointcut designator, yet this simple example already introduces a problem the implementer of the extension should be aware of. The keyword `cast` is reserved in the context of a pointcut, which means that it is no longer allowed as part of a name pattern (see Section 5.6.2). To resolve this, the keyword should be added to the *simple name patterns* explicitly, which has not been done for this extension in the abc implementation. The same problem occurs in the implementation of global pointcuts 59 (a mechanism for globally restricting some aspects by extending their pointcut definitions). In our syntax definition this is not an issue, since the keywords are reserved per nonterminal.

```
module AspectJMix[Ctx]
imports AspectJ
  [ AspectDec     => AspectDec[[Ctx]]
    AspectBodyDec => AspectBodyDec[[Ctx]]
    ...
    TypePattern   => TypePattern[[Ctx]]
    PointcutExpr  => PointcutExpr[[Ctx]] ]
```

Figure 5.19 Grammar Mixin for AspectJ

CFlow Level

The `cflowlevel` [10] pointcut designator is an extension used to select join points based on the level of recursion. The `cflowlevel` pointcut designator takes two arguments: a number for the recursion level and a pointcut. However, the lexical state for pointcuts in `abc` does not allow integer literals. To avoid the need for a new lexical state or other complex solutions, the syntax of the `cflowlevel` construct was changed to a string literal, which is supported in the pointcut lexical state [11]. Unfortunately, in this case the syntax of the extension was designed to fit the existing lexical states of the scanner. In the SDF implementation of this extension referring to an integer literal is not a problem.

5.9.3 *Open Modules*

Open modules were proposed by Aldrich [Aldrich 2005] to solve the coupling issues that arise between aspects and the code they advise. It provides an encapsulation construct that allows an implementation to limit the set of points to which external advice applies. Recently, an `abc` extension was proposed that extends open modules to full AspectJ ([Aldrich 2005] deals with a small functional language) and defines appropriate notions of module composition [Ongkingco et al. 2006]. The normal form of open modules as proposed in [Ongkingco et al. 2006] is as follows:

```
module ModuleName {
   class class name pattern
   friend list of friendly aspects
   expose : pointcut defining exposed join points
}
```

A module declaration applies to a set of classes as specified in the `class` part. It states that aspects can only advise join points matched by the pointcut specified in the `expose` part. *Friendly aspects*, listed in the `friend` part, have unrestricted access to the join points occurring within classes of the module. The exact syntax is more elaborate for notational convenience, and also includes constructs for restricting or opening modules upon composition.

The parsing of open modules requires a new lexical state. This need falls out of the designed extensibility of `abc`, as highlighted in Section 5.6. As a

[10] Available at `http://www.cs.manchester.ac.uk/cnc/projects/loopsaj/cflowlevel/`
[11] See http://abc.comlab.ox.ac.uk/archives/dev/2005-Aug/0003.html

```
module OpenModule[JavaCtx]
exports
  context-free syntax
    ModDec+ -> CompilationUnit[[JavaCtx]]
    Root? "module" Id "{" ModMember* "}" -> ModDec
    "class"     ClassNamePattern ";"  -> ModMember
    "friend"    {AspectName ","}+ ";" -> ModMember
    "open"      {Module ","}+ ";"     -> ModMember
    "constrain" {Module ","}+ ";"     -> ModMember
    Private? "expose"    ToClause? ":" PointcutExpr ";" -> ModMember
    Private? "advertise" ToClause? ":" PointcutExpr ";" -> ModMember
    "to" ClassNamePattern -> ToClause

lexical syntax
  "root" | "module"            -> Keyword[[JavaCtx]]
  "module" | ... | "advertise" -> Keyword
```

Figure 5.20 SDF module extending AspectJ with open modules.

consequence the full scanner has to be copied and modified. Although just 15 lines of code had to be modified in the copy, this introduces a maintenance problem: copying the scanner implies that the developer of the extension has to keep the extension in sync with the main scanner of abc, which is bound evolve, for example to introduce support for AspectJ 5.

Conversely, SDF allows the syntax of open modules to be concisely and modularly expressed, as illustrated in Figure 5.20. A new context can be introduced in a *modular* way. The implementation is based on the AspectJ grammar mixin module of Figure 5.19.

5.9.4 *Context-Aware Aspects*

We now consider the AspectJ syntax extensions for the pointcut restrictors proposed in [Tanter et al. 2006] for *context-aware aspects*. Context-aware aspects are aspects whose pointcuts can depend on external *context* definitions, and whose advices may be parameterized with context information. Contexts are stateful, parameterized objects: they are specified by implementing a context class with a method that determines whether the context is active or not at a given point in time; context activation can be based on any criteria, like the current control flow of the application, some application-specific condition, or input from environment sensors. A context is an object that may hold relevant state information (such as the value obtained from a given sensor).

Context-aware aspects [Tanter et al. 2006] propose a number of general-purpose and domain- or application-specific pointcut restrictors for restricting the applicability of an aspect based on some context-related condition. These pointcut restrictors are explained using an AspectJ extended syntax, although only a framework-based implementation is provided, based on the Reflex AOP kernel [Tanter & Noyé 2005].

The `inContext` pointcut restrictor is similar to an `if` pointcut designator, restricting the applicability of an aspect (e.g. `Discount`) to the application currently being in a certain context (e.g. `PromotionCtx`):

```
pointcut amount():
  execution(double Item.getPrice()) && inContext(PromotionCtx);
```

Also, context-aware aspects provide a mechanism to expose state associated to the context (e.g. a discount rate) as a pointcut parameter, subsequently it can be used in the advice. In the following example, the `rate` property of the `PromotionCtx` is exposed in the pointcut and subsequently used in the advice to compute the associated discount.

```
aspect Discount {
  pointcut amount(double rate):
    execution(* ShoppingCart.getAmount())
        && inContext(PromotionCtx(rate));

  double around(double rate): amount(rate) {
    return proceed() * (1 - rate);
  }
}
```

Context activation can be parameterized in order to foster reuse of contexts. For instance, a stock overload context can be parameterized with the ratio of stock overflow required to be considered active. In the following example, the `amount` pointcut matches only if the stock overload factor is superior to 80% when the rest of the pointcut matches.

```
pointcut amount():
  execution(* ShoppingCart.getAmount())
      && inContext(StockOverloadCtx[.80]);
```

An important characteristic of the approach presented in [Tanter et al. 2006] is the possibility to extend the set of pointcut restrictors, either general purpose or domain/application specific. Hence the set of context restrictors is *open-ended*. An example of a general-purpose restrictor is one that makes it possible to refer to past contexts, such as the context at creation time of an object. For instance, the `createdInCtx` restrictor in the next example refers to the context in which the currently-executing object *was* created. The `amount` pointcut matches if the current shopping cart was *created* in a promotional context, independently of whether the promotion context is still active at check-out time.

```
pointcut amount():
  execution(* ShoppingCart.getAmount()) && createdInCtx(PromotionCtx);
```

An example of application-specific restrictor is `putInCartInCtx`, which refers to the context at the time an item was put in the shopping cart:

```
pointcut amount():
  execution(* Item.getPrice()) && putInCartInCtx(PromotionCtx);
```

```
module CtxAspect
exports
  context-free syntax
    "inContext" "(" ActualCtx ")"    -> PointcutExpr
    "createdInCtx" "(" ActualCtx ")" -> PointcutExpr
    TypeName[[JavaCtx]] ACParams? ACValues? -> ActualCtx
    "[" {Expr[[JavaCtx]] ","}+ "]" -> ACParams
    "(" {CtxId[[JavaCtx]] ","}+   ")" -> ACValues

  lexical syntax
    "inContext" | "createdInCtx" -> Keyword[[PointcutCtx]]
```

```
module EShopCtxAspect
imports CtxAspect
exports
  context-free syntax
    "putInCartInCtx" "(" ActualCtx ")" -> PointcutExpr

  lexical syntax
    "putInCartInCtx" -> Keyword[[PointcutCtx]]
```

Figure 5.21 Two SDF modules for context-aware aspects: (top) general-purpose pointcut restrictors; (bottom) application-specific extension for the EShop.

Parsing Context-Aware Aspects

Extending AspectJ with the two general-purpose context restrictors `inContext` and `createdInCtx` can be defined in a `CtxAspect` SDF module (Figure 5.21 (top)). The context-free syntax section defines the new syntax: a context restrictor followed by the actual context definition; a context is a Java type name, with optional parameters and values (for state exposure). The lexical syntax section specifies that the new pointcut restrictors have to be considered as keywords in a pointcut context.

Figure 5.21 (bottom) shows a modular syntactic extension for context-aware aspects with the definition of the `putInCartInCtx` application-specific restrictor. Interestingly, it is not necessary to redefine the syntax for parameters and values in the new syntax extension definition (`ActualCtx` is visible from `EShopCtxAspect`).

5.10 PERFORMANCE

Deriving a production quality (i.e. efficient and with language-specific error reporting) parser from a declarative, possibly ambiguous, syntax definition is one of the open problems in research on parsing techniques. In particular, the area of scannerless parsing is relatively new and the number of implementations is very limited (i.e. about 2). This work does not improve the performance, error reporting or error recovery of these parsers in any way: besides the arguments for a declarative specification of AspectJ, it only provides a strong motivation for continued research on unconventional parsing

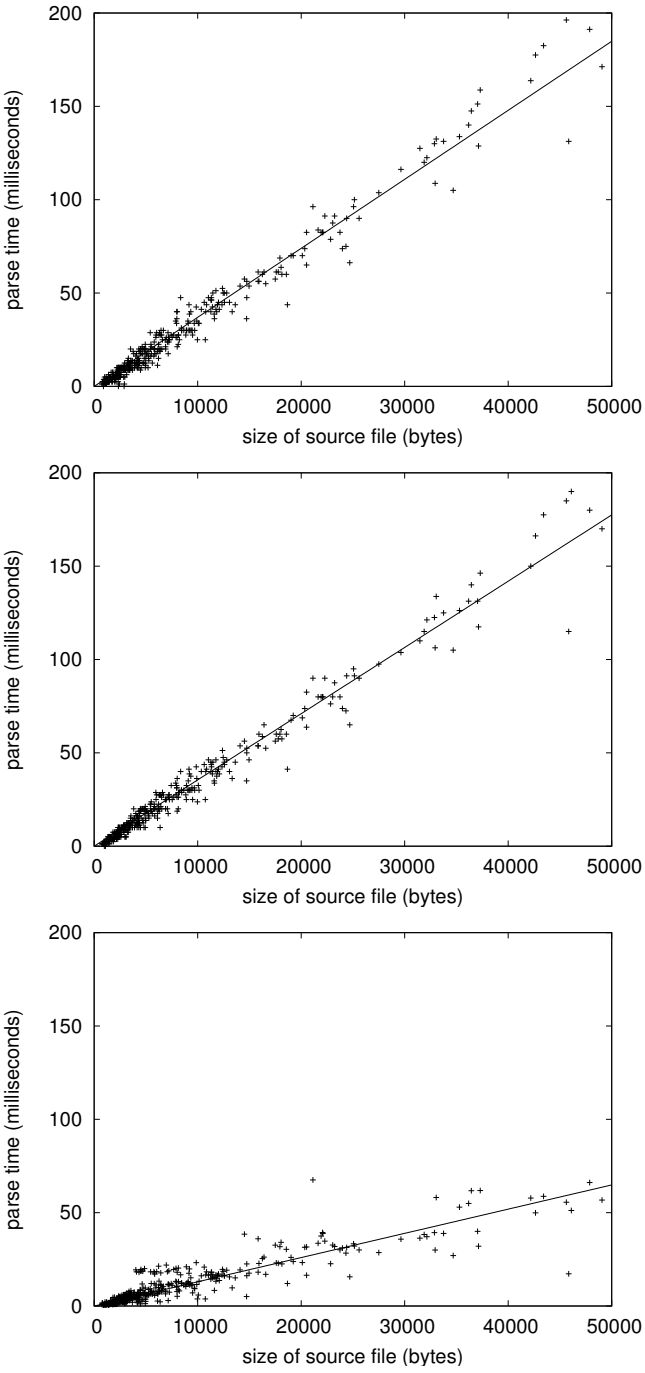

Figure 5.22 Benchmark of parsing Java source files. Top to bottom: sglr/ajc, sglr/java, and abc

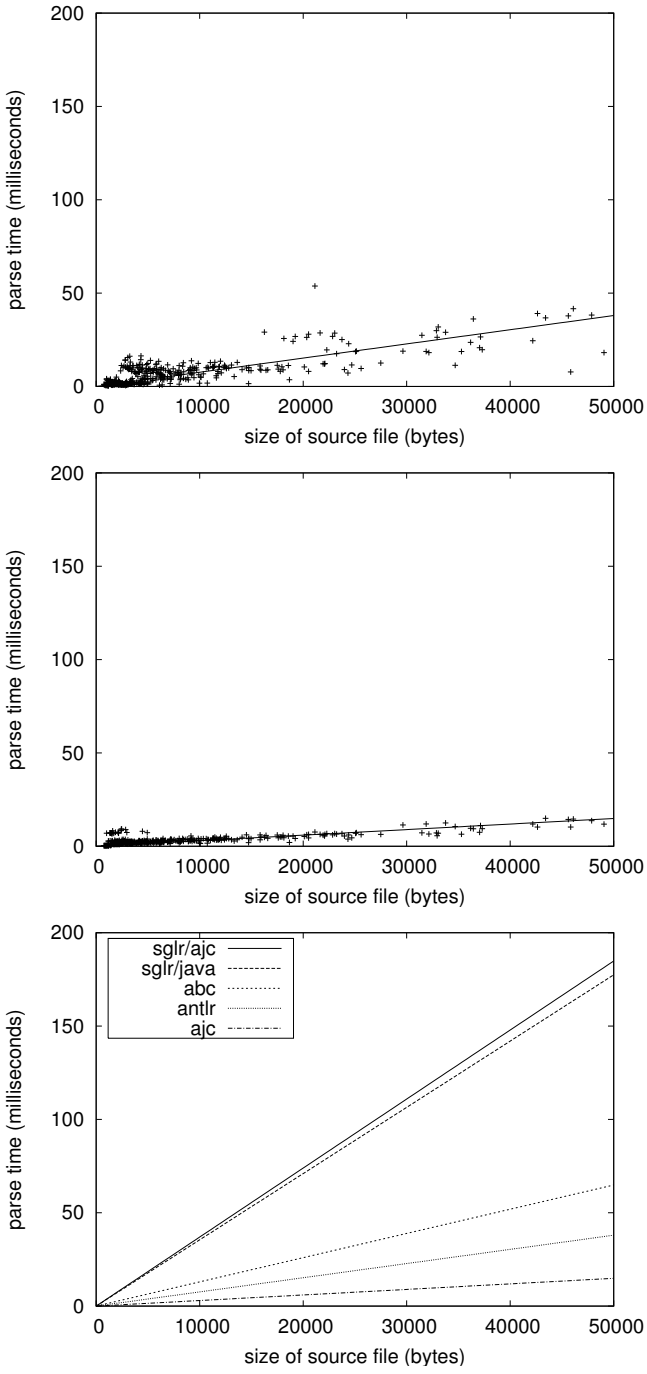

Figure 5.23 Benchmark of parsing Java source files. Top to bottom: ANTLR, `ajc`, and the trend lines of all the benchmarked parsers in a single graph

techniques. Although our current objectives are not to replace every single parser in a production compiler by a scannerless generalized LR parser, it is good to get an impression of the current state of a scannerless generalized LR parser compared to parsers used in existing compilers.

In order to evaluate the applicability of our approach beyond specification purposes, we have performed some benchmarks to estimate the efficiency of scannerless generalized LR parsing. It has been shown that although $O(n^3)$ in the worst case (with n the length of the input), generalized LR performs much better on grammars that are near-LR [Rekers 1992], and that the cost of scannerless parsing is linear in the length of the input, although with an important constant factor [Salomon & Cormack 1989]. There is little knowledge of how the integration of scannerless and generalized LR parsing performs. We hereby compare the cost of the SGLR parser with that of `abc`, `ajc`, and ANTLR [Parr] (an LL(k) parser generator) when parsing both a massive amount of Java code and the AspectJ testsuite of `abc`.

5.10.1 *Benchmark Setup*

The test machine is an Intel Pentium 4 3.2GHz CPU with 1GB memory, running SUSE 9.0. The `abc`, `ajc`, and ANTLR parsers use the Sun JDK 5.0. SGLR 3.15 is invoked with heuristic filters and cycle detection disabled. For all parsers, we only measure the actual parse time: this includes the construction of the parse tree, but no semantic analysis and I/O costs. In all benchmarks, the same source file is parsed 15 times and the first 10 parses are ignored to avoid start-up overhead (class loading and JIT compilation for Java, parse table loading for SGLR). For ANTLR we use version 3.0b3 and a recent Java 1.5 grammar written by Terence Parr.

5.10.2 *Benchmark Results*

Figure 5.23 shows the results of the Java benchmark: parsing of the source files of the Azureus Bittorrent client and Tomcat 5.5.12. This figure shows that parsing with all parsers is linear in the size of the input, illustrated by the trend lines (calculated using least-squares). SGLR parsing with the AspectJ grammar is about 4% slower than parsing with the Java grammar. The constant factor of parsing with `abc` is about 40% of the factor of SGLR. Clearly, the performance of `ajc` is superior to all the other parsers. The performance of the ANTLR Java parser is more or less between the `abc` and `ajc` parsers, but this is a plain Java parser. The creation of ANTLR parsers has been heavily optimized in this benchmark after noticing the substantial setup cost of ANTLR3 parsers. The absolute times are all fractions of second, which is only a very small portion of the total amount of time required for compiling an AspectJ program, since the most expensive tasks are in semantic analysis and actual weaving of aspects.

Figure 5.24 shows the results for parsing aspect code from the testsuite of `abc`. Note that the scales are different, since aspect sources are typically

smaller. Again, the parse time is linear in the size of the input, but the constant factor of `abc` is about 60% of the factor of SGLR. The performance of SGLR compared to `ajc` has improved as well. For both Java parsers, parsing source files close to 0 bytes is relatively expensive. The reason for this is JIT compilation, which still introduces start-up overhead after parsing the same file 10 times before the actual benchmark. At first, we ignored just the first two parses, which had a dramatic impact on the performance. Overall, the parse time is always smaller than 0.06 second, so the absolute differences are extraordinarily small for these tiny source files. We would have to benchmark larger aspect sources (which do not exist yet) to get more insight in the performance of parsing aspects and pointcuts.

As a matter of fact, a typical project consists of a lot of Java code with a few AspectJ aspects, so the Java benchmark is particularly relevant. To conclude, the absolute and relative performance of scannerless generalized LR parsing is promising for the considered grammars (Java and AspectJ). The fact that the parsers are implemented in Java versus C is not relevant, since the most important question is whether SGLR is fast enough in absolute time. Nevertheless, since there is virtually no competition in the area of scannerless parsing at present, there is ample opportunity for research on making the performance of scannerless parsing even more competitive.

5.10.3 *Testing*

The compatibility of the AJC syntax definition is tested heavily by applying the generated parser to all the valid source files of the testsuite of `ajc` 1.5.0. Testing invalid sources requires the examination of the full `ajc` testsuite to find out if tests should fail because of semantic or syntactic problems. This is a considerable effort, but will be very useful future work. The results of the testsuite are available from the web page mentioned in the introduction.

5.11 DISCUSSION

5.11.1 *Previous Work*

Although SDF has a long history [Heering et al. 1989], a more recent redesign and reimplementation as SDF2 [Visser 1997b, van den Brand et al. 2002] has made the language available for use outside of the algebraic specification formalism *ASF+SDF*. This redesign introduced the combination of scannerless and generalized LR parsing [Visser 1997a].

In [Bravenboer & Visser 2004 (Chapter 2)] we motivated the use of SGLR for parsing embedded domain-specific languages. This method, called MetaBorg, focuses on creating new language combinations, where it is important to support combinations of languages with a different lexical syntax. In [Bravenboer et al. 2005 (Chapter 3)] we presented the introduction of *concrete object syntax* for AspectJ in Java as a reimplementation of the code generation tool Meta-AspectJ [Zook et al. 2004]. In that project, we used the AspectJ syntax

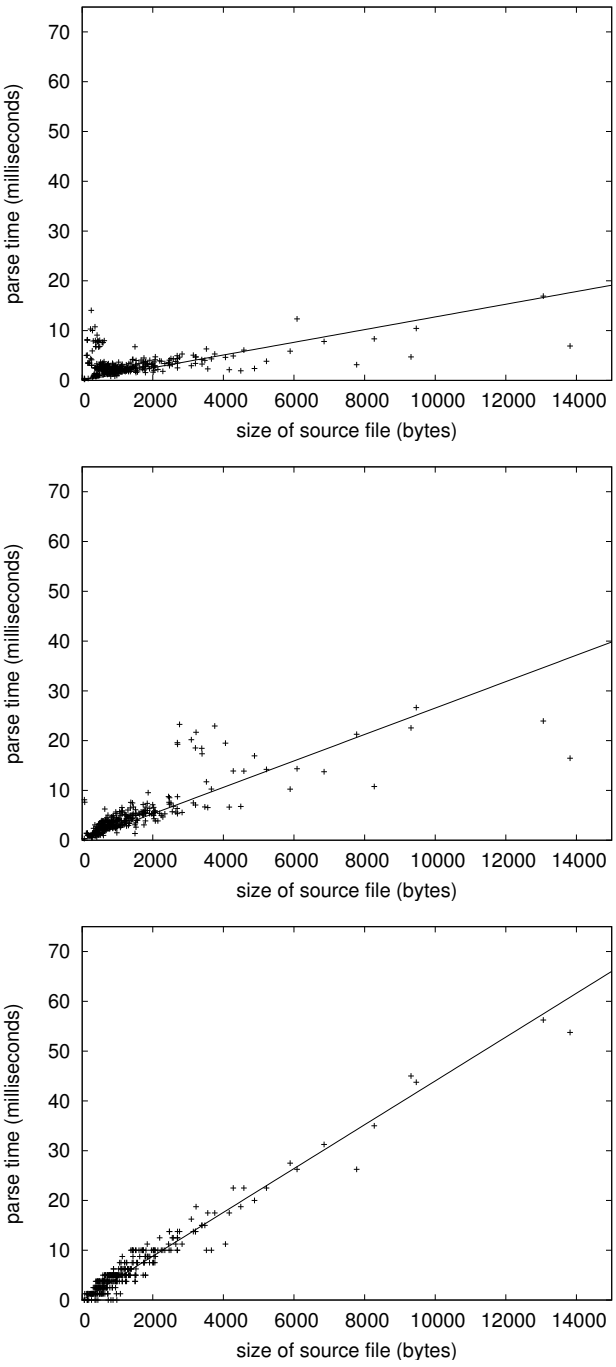

Figure 5.24 Benchmark of parsing AspectJ source files. Top to bottom: `ajc`, `abc`, sglr/ajc

definition of this chapter, but the design and benefits of the grammar were not discussed.

Compared to this earlier work, we have discussed the design of the syntax definition of AspectJ, which poses a challenge to parser generators. We discussed in detail how complex differences in lexical syntax of the involved languages can concisely be defined in SDF, and how this is related to a stateful lexer. The syntax definition for AspectJ provides a compelling example of the application of scannerless parsing to an existing language, where all of the following features of SDF prove their value: modular syntax definition, rejects for keywords, scannerless parsing, generalized LR parsing, parameterized symbols, and parameterized modules. Grammar mixins are a surprisingly useful application of parameterized modules and non-terminals. Also, we presented a solution for implementing context-sensitive keywords in SDF.

5.11.2 *Related Work*

Using a scannerless parser for AspectJ has not been proposed or even mentioned before in the literature. Concerning AspectJ implementations, we have discussed the `abc` and `ajc` scanners and parsers at length.

The advantage of separate scanners is their foundation on finite automata, which allows fast implementations. However, this characteristic also implies obliviousness to context, while processing languages such as AspectJ requires introducing context-sensitivity into scanners. There are essentially two options to make scanners context-sensitive. First, the scanner may interact with the parser to retrieve the context of a token. This is not very difficult to implement in a handwritten parser, but parser generators based on this approach are rare, and as far as we know none have been used to parse AspectJ. Blender [Begel & Graham 2004] uses GLR parsing with an incremental lexical analyzer that is forked together with the LR parsers in the GLR algorithm. A similar approach was also used by the implementation of SDF before the introduction of scannerless parsing [Heering et al. 1989]. Second, lexical analysis may be extended with a rudimentary form of context-free parsing to recognize the global structure of the source file while scanning by means of *lexical states*, without interaction with the parser. This approach is used in the `abc` scanner and parser for AspectJ.

DURA-LexYt [Blasband 2001] supports lexical analysis with multiple interpretations of the input. As opposed to scannerless parsing, a separate scanner with support for backtracking is used. In this way, choosing the correct lexical interpretation can be controlled by the parser without the need for managing lexical states in the scanner. DURA provides several levels of lexical backtracking to facilitate typical scenarios of tokenization (e.g. a single or multiple divisions into tokens), whereas scannerless parsing requires this to defined explicitly. Lexical backtracking can be used for context-sensitive lexical analysis, but does not facilitate context-specific reserved keywords. DURA-LexYt does not support inherent ambiguities: the parser always returns a single parse tree, which might also not be the one desired. More experience with

lexical backtracking is required to get insight in the performance compared to scannerless parsing.

JTS Bali [Batory et al. 1998] is a tool for generating parsers for extensions of Java. It supports composition of lexical syntax based on heuristics such that the best results are produced in the common cases. For example, keyword rules are put before the more general rules, such as for identifiers. This means that it cannot handle lexical state and it not suitable for defining AspectJ-like extensions of Java.

Parsing techniques with higher-order (parameterization) features, such as parser combinators in higher-order functional languages [Hutton 1992], allow reuse and abstraction over grammars, but do not support *unanticipated* reuse of a grammar. Grammar mixins, on the other hand, are modules based on unparameterized grammars (e.g. Java) that make this grammar reusable and allow unanticipated modification of the grammar in every context.

5.11.3 *Future Work*

Grammar Mixins

In this chapter we have applied grammar mixins and explained their functionality only in an informal way. In future work, we plan to make the notion of grammar mixins more formal. In particular, the semantics of mixin composition of grammars that already use mixins itself needs to be defined more precisely. Also, grammar mixins should be integrated in a syntax definition formalism. Currently, an external tool is used to generate grammar mixin modules, which is not desirable. Furthermore, a notion of interfaces for grammar mixins would be useful to separate the implementation of a mixin from its interface. Finally, multiple instantiations of a grammar mixin for a relatively large language, such as Java or AspectJ has a major impact on the performance of the parser generator, which could again be solved by integration of grammar mixins in the syntax definition formalism. Chapter 6 presents a first ingredient of this solution: *parse table composition*. Parse table composition enables separate compilation of grammars involved in a language conglomerate. The grammar of Java is compiled to a parse table component that can be instantiated multiple times in an efficient way. In this way, we avoid the application of the parser generator to all the separate instantiations.

Improvements to SDF and SGLR

As we have shown in this chapter, SDF provides a declarative approach to solving complex parsing problems. Yet, the formalism and tools are not in widespread use. What may be the reason for this (other than publicity) and what improvements can be made?

Rule Syntax SDF's reverse grammar production rules may make developers accustomed to BNF style rules uncomfortable. It might make sense to provide a version of SDF using such a conventional style.

Performance The benchmarks showed that the performance of the SGLR parser is a constant factor slower than the abc parser, which should be acceptable for use at least in research projects. However, there is good hope that the performance of SGLR can be much improved. There are alternative GLR implementations (e.g. [Aycock & Horspool 1999, McPeak & Necula 2004]) and alternative algorithms such as right-nulled GLR [Scott & Johnstone 2006] with better performance than SGLR. However, these techniques have not yet been extended to scannerless parsing, while scannerlessness is essential in our syntax definition for AspectJ. Even after these techniques are adopted, there remains a theoretical performance gap between GLR and LALR, since the complexity of GLR depends on the grammar. Therefore, it would be useful to develop profiling tools that help grammar developers to detect performance bottlenecks in grammars.

Error Reporting The current error reporting of SGLR is rather Spartan; it gives the line and column numbers where parsing fails. This may be improved using a technique along the lines of the Merr tool that generates error reporting for LR parsers from examples [Jeffery 2003]. This requires an adaption of the techniques where the set of parsing states at the failure point is interpreted.

Analyzing Ambiguities The disadvantage of LR-like parser generators is that the grammar developer is confronted with shift-reduce and reduce-reduce conflicts. However, this is also their advantage; the developer is forced to develop an unambiguous grammar. When using GLR there is no need to confront the developer, however, the conflicts are still there to inspect. It would be useful to develop heuristics that can be used to inspect the conflicts in the parse table and use these to point the developer to problematic parts in the grammar.

Platform A more mundane, not so scientific reason for lack of adoption may be the platform. The SDF parser generator and the SGLR parser are implemented in C and the distribution is Unix/Linux style. Furthermore, parse trees and abstract syntax trees are represented using ATerms, which requires linking with the ATerm library. Retargeting the SDF/SGLR implementation to other platforms, such as Java, may help adoption.

Applications of the AspectJ Syntax Definition

With respect to the AspectJ syntax definition itself, there are a number of applications to consider.

AspectJ Specification For widespread acceptance of aspect-oriented languages, a complete specification of the syntax and semantics of the language is important. In particular, concerns about modifying the semantics of the host language could be reduced by at least having a complete specification of the syntax of the language. If there is enough interest in the specification of the syntax and semantics of the AspectJ language, then we would like to work

with the AspectJ developers to make the current syntax definition even more compatible with `ajc` and make it the basis of such a specification.

As one of the first applications, the `abc` team has used our syntax definition of AspectJ in a definition of the semantics of static pointcuts, defined as a set of rewrite rules from AspectJ pointcuts to Datalog [Avgustinov et al. 2007].

Connecting to the AspectBench Compiler Considering the extensibility goals of `abc`, our modular and extensible definition of AspectJ would be most useful as part of the front-end of `abc`. Also, we have shown that pseudo keywords do not require a handwritten parser, so the `abc` compiler could be made more compatible with syntax accepted by `ajc`.

Multi-language AOP We are currently working on integrating the MetaBorg approach [Bravenboer & Visser 2004 (Chapter 2)] and the Reflex AOP kernel project [Tanter & Noyé 2005] for multi-language AOP. The current AspectJ syntax definition can be used to support AspectJ in Reflex, allowing AspectJ extensions to be prototyped conveniently.

5.12 CONCLUSION

We have presented the design of a modular syntax definition for the complete syntax of AspectJ, i.e. integrating the formalization of the lexical and the context-free syntax of the language. In addition, we have shown that scannerless parsing in combination with an expressive module system can elegantly deal with the context-sensitive lexical syntax of AspectJ. The result is a syntax definition that achieves a new level of extensibility for AspectJ, which is useful for research on aspect-oriented programming extensions. The performance of the scannerless generalized LR parser for this grammar turns out to be linear with an acceptable constant factor, which opens up possibilities for the integration of our solution in extensible compilers for AspectJ.

Furthermore, our work on syntax definition for AspectJ provides guidelines for approaching the current trend to design programming languages that are in fact mixtures of various sublanguages, for example for the integration of search capabilities or concrete object syntax (e.g. LINQ, E4X, XQuery, Cω). The convention of separating the parsing process into a scanner and a parser does not apply to such languages, requiring language designers and implementers to reconsider the parsing techniques to use.

With the syntax definition for AspectJ, we have shown that scannerless generalized LR parsing is not just useful for reverse engineering, meta programming, interactive environments, language prototyping, and natural language processing, but that scannerless generalized LR may at some point be used in compilers for modern general-purpose languages. AspectJ makes a strong case for the use of scannerless parsing to provide concise, declarative specification and implementation of the next-generation of programming languages.

This result provides a strong motivation for addressing the barriers to a wider adoption of scannerless generalized LR parsing that we observed in the previous section.

ACKNOWLEDGEMENTS

At Utrecht University this research was supported by the NWO/Jacquard project TraCE (638.001.201). Éric Tanter is partially financed by the Millenium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile. We thank the abc team for the report on the abc scanner and parser. The description of lexical states was very useful in the development of our syntax definition. Pavel Avgustinov of the abc team provided extensive feedback on the syntax definition. We thank Arthur van Dam for his help with Gnuplot, and Jurgen Vinju for his advice on benchmarking SGLR. We thank Mark van den Brand, Jurgen Vinju and the rest of the SDF/SGLR team at CWI for their work on the maintenance and evolution of the SDF toolset. We thank Rob Vermaas for his help with the implementation of the syntax definition and benchmarking the generated parser. Finally, we would like to thank Eelco Dolstra, the anonymous reviewers of CC 2006, and the anonymous reviewers of OOPSLA 2006 for providing useful feedback on earlier versions of this chapter.